Universität Bielefeld

**Forschungsberichte der
Technischen Fakultät
Abteilung Informationstechnik**

Fundamental Algorithms for a Declarative Pattern Matching System

Stefan Kurtz

Report 95-03

Universität Bielefeld • Postfach 10 01 31 • 33501 Bielefeld • FRG

Impressum: Herausgeber:
Robert Giegerich, Alois Knoll, Peter Ladkin,
Helge Ritter, Gerhard Sagerer, Ipke Wachsmuth

Technische Fakultät der Universität Bielefeld,
Abteilung Informationstechnik, Postfach 10 01 31,
33501 Bielefeld, FRG

Fundamental Algorithms for a Declarative Pattern Matching System

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Universität Bielefeld

vorgelegte

Dissertation

von

Stefan Kurtz

Bielefeld, im Juli 1995

Acknowledgments

I would like to thank my supervisor Robert Giegerich for his valuable interest, guidance and encouragement. I am indebted to Enno Ohlebusch who has patiently given me much of his time to discuss numerous topics of the thesis. Special thanks go to Esko Ukkonen for helpful discussions during his visit in Bielefeld. Valuable contributions to the quality of the thesis were made by Dorothee Berndt and Jens Stoye. I would like to thank Bernd Bütow, Georg Füllen, Frank Hischke, Antje Krause, Marc Rehmsmeier, and Karsten Loer for carefully reading parts of the thesis. Finally, I am much obliged to Preston Crutchfield and Birgit Kurtz for improving my English.

Contents

Chapter 1

Motivation and Overview

String Algorithms in the Functional Programming Paradigm

A large part of human knowledge is collected in the form of strings of symbols over some suitable alphabet. Strings have to be efficiently processed in many different ways. Some operations on strings, like string searching and string comparison, are so important in any processing of information that they can be considered important primitives of computation.

In the last 15 years, the motivation for research in string algorithms came to a substantial part from several application fields like information retrieval, computer vision, and molecular biology. Especially in the latter field, a large number of interesting string searching and comparison problems appeared in the context of analyzing DNA, RNA, and protein sequences. This led to the development of a multitude of software, ranging from programs solving isolated problems [LP85, AGM+89] to systems providing a large number of functions for simple sequence analysis [DHS84, Sta88, Ger91]. To cope with the exponential growth of biological sequence data (to the current volume of about 100 million nucleotides), the primary design goal for these tools was efficiency. Only rudimentary ways to combine the basic functions were incorporated. For a more complex analysis, it was necessary to resort to the operating system level.

Newer developments reacted upon this severe drawback. In [Sea89, Meh91, Gon92, MM93b], different experimental systems were proposed that allowed for a convenient description of complex patterns, and incorporated several novel ideas to efficiently search for these patterns. However, all systems had a common disadvantage. They could not easily be adjusted to new requirements. That is, the user was in general not able to integrate his own functionality with built-in features. This was because the systems were only extensible (if at all) on a very low level of abstraction which is known as error-prone and unproductive.

Recently, Giegerich [Gie92] suggested to embed a tool for sequence analysis in the functional programming paradigm. In particular, he described a declarative pattern matching system named *Pamelita*,[1] which provided several basic matching functions and utilities to combine them to complex matchers. Building upon this machinery, the system can be extended on a coherent level of abstraction. More precisely, the user can obtain a family of matching functions tailored to a specific biological research context by providing some simple functional

[1] The name *Pamelita* was chosen since the system mimics the PAMALA system of Mehldau [Meh91].

programs. This idea is very promising because it could gain significantly from the widely known virtues of functional programming, such as transparency, compositionality, type-safeness and high programming productivity.

Pamelita is more a research program than a prototype. To become a useful tool for biological sequence analysis, there are many improvements necessary. In [Gie92], Giegerich enumerates ten work items, most of which concern language design and user interfaces. However, the most important point is to provide a large number of efficiently implemented basic matching functions. In other words, string searching and string comparison algorithms have to be embedded in the functional programming paradigm. A first step addressing this demand was made in [Gie92] where the construction of suffix trees and the search for repeat structures was studied. This was done in an ad hoc manner, but the results were promising and encouraged Giegerich and Kurtz [GK94] to have a closer look especially at functional suffix tree constructions.

In the thesis we will consider the embedding of fundamental string algorithms in the functional programming paradigm more systematically and in greater detail. In particular, we provide a thorough development and complete implementation of several string searching and comparison algorithms. We mainly restrict ourselves to well established algorithms, but we also provide some new algorithmic ideas.

Unlike traditional imperative presentations, we apply the structuring methods of functional languages to construct implementations from individual reusable components. The use of higher order functions and lazy evaluation allows to "glue" program components together in ways that are not supported by modular imperative languages like Modula-2 and Ada. By this method we obtain a greater degree of modularity than usually seen.

Although a *Pamelita*-like tool will be an important application for our functional implementations, the use is not restricted to problems occurring in the analysis of biological sequences. Due to their flexibility and modularity, our implementations form a powerful collection of functions to be conveniently reused in a variety of contexts. For example, we have used this collection several times for rapid prototyping of string algorithms. Some students at the University of Bielefeld [Ber95, Kra95, San95, Sto95] have already based parts of their master's thesis on this collection.

Heretofore, string algorithms have always[2] been given in an imperative style (see the textbooks [Sed88, CLR90, GBY91, BY92, Ste94]) with asymptotic efficiency analysis for the random access machine [AHU74]. Instead, we use functional style and analyze the efficiency in the computation model of outermost graph reduction. The implementation methods for lazy functional languages (see [PJ87, PJL91]) usually correspond more or less closely to this model, so that it is generally a good predictor of the behavior of an actual implementation. However, we will also validate the analytical efficiency results by practical measurements.

Note that our implementations are more than a reimplementation of known algorithms. Since the programming techniques in the imperative and the functional world are often incompatible, our implementations often look quite different from their imperative counterparts, even if they are based on the same algorithmic ideas. One of the main contributions of the thesis is to show that despite these differences the functional programs can (except for one case) compete with the corresponding imperative programs concerning asymptotic

[2] We only know of the three exceptions [All92, SR92, GK94] which use functional style.

efficiency. This result is remarkable since it has been questioned that purely functional programs can achieve the same complexity class as imperative programs.

The dissimilarity of imperative and functional style has an important consequence for the structure of the thesis. Rather than referring to some well-known operational description in form of an imperative program, we view a problem from a declarative point and develop an algorithm from its basic idea. In particular, we will establish the properties that are responsible for the correctness and efficiency of the algorithm. This also includes a careful analysis of the boundary cases, in order to achieve robust implementations. This process simplifies and clarifies the structure of an algorithm and leads to a declarative description which does not refer to a particular computation model. Note that even for well-studied problems, a declarative view sometimes leads to interesting insights. Giegerich and Kurtz [GK95a] have demonstrated this with the development of the lazy suffix tree construction. We will give more examples in the thesis.

The declarative description is a very important step in the program development. It specifies an algorithm on an abstract level, but it is precise enough to argue about its efficiency aspects. Moreover, it often maps to a functional implementation in a transparent and relatively straightforward way.

Note that with respect to the functional programs, the thesis is self-contained. For all functions that we use in the implementations, the complete and real code is given. Indeed, the text of the thesis is an executable and extensively tested collection of functional programs with "comments" in LaTeX format.

Structure of the Thesis and Summary of the Results

Chapter 2 is devoted to functional programming concepts. First, we recall the fundamental differences between the imperative and the functional programming paradigm. Then we shortly discuss the most important properties of functional languages including some aspects of different evaluation strategies. For those readers without experience in the functional world, we introduce the basic ideas and the most important notations of the purely functional programming language Miranda[3] [Tur85]. This is done in an informal way by the use of several examples. We restrict ourselves to a subset of Miranda which suffices for most of the implementations given in the thesis. In section 2.2, we describe the model of computation according to which we determine the efficiency of our functional programs. Subsequently, we review Wadler's [Wad90, Wad92b, Wad92a] monadic programming style, and outline how it can be used to incorporate updatable arrays in Miranda without destroying the referential transparency of the language. Finally, we establish two extensions of Miranda that are necessary for some of the implementations that we give. The first extension provides arrays with an efficient creation and lookup function. The second extension additionally includes an efficient array update function.

In chapter 3, we describe our string processing machinery. We begin with some basic definitions and notations. Section 3.2 introduces the concept of $\mathcal{A}^+$-trees which occurs in many variants throughout the thesis: $\mathcal{A}^+$-trees serve as the "raw material" for suffix trees, and they are used to implement the Knuth-Morris-Pratt Algorithm [KMP77] and the Aho-Corasick

[3] Miranda is a trademark of Research Software Limited.

Algorithm [AC75]. The main contribution of section 3.2 is the notion of locations which allows to describe algorithms on $\mathcal{A}^+$-trees in a convenient and concise way. Section 3.3 is devoted to suffix trees. We outline some applications, consider space requirements, and explain the role of the sentinel character. In section 3.4, we regard implementation issues. In particular, we discuss some important design decisions concerning the representation of strings, edge labels, and edge sets. To cope with the different degrees of compactness and specific annotations, we introduce a polymorphic data type for $\mathcal{A}^+$-trees. Moreover, we show how to implement some basic functions on locations to be used throughout the thesis. Section 3.5 is devoted to suffix tree construction. We begin with the lazy suffix tree algorithm. Its worst case efficiency is quadratic, but it leads to a simple functional implementation with a good average case performance. We also describe Ukkonen's linear online algorithm [Ukk93b] and the widely known linear algorithm of McCreight [McC76]. All three suffix tree algorithms are discussed in a similar way in [GK95a]. However, we use a slightly different notation and make some technical details explicit which are left open in [GK95a]. This leads to a more compact and concise declarative description which provides the basis for proving the correctness of the algorithms. At least for Ukkonen's algorithm, we give the first detailed correctness proof. Section 3.6 shows the functional implementation of the three suffix tree algorithms. Since Ukkonen's and McCreight's algorithms require updatable states, we implement them in monadic programming style. In this way, we obtain the first purely functional implementation of linear suffix tree algorithms. The main contribution of section 3.7 is a higher order function to compute suffix trees including suffix links and annotations in a unified way. In section 3.8, we briefly recall the concept of deterministic finite automata and show how to implement these purely functionally using circular structures. Section 3.9 is devoted to string comparison models. In particular, we consider the edit distance model [Ula72], the maximal matches model [EH88], and the q-gram model [Ukk92a]. We describe the Wagner-Fischer Algorithm [WF74] which uses the technique of dynamic programming to compute the edit distance. An important contribution is a new algorithm which performs a parallel walk of two suffix trees in order to compute the q-gram distance. In section 3.9.4, we briefly discuss the significance of the three comparison models in molecular biology. Section 3.9.5 is devoted to implementation issues. We provide a higher order function that allows for the implementation of many variations of the Wagner-Fischer Algorithm in a convenient and purely functional way. Moreover, we present a new and very flexible approach for computing suboptimal alignments.

In chapter 4, we consider the exact string searching problem which consists of finding exact occurrences of a pattern in an input string. We present the most important solutions to the problem and show how to implement them in our functional framework. All functional implementations have the same asymptotic efficiency as their imperative counterparts. We begin with the Brute Force Algorithm. Subsequently, we consider the Knuth-Morris-Pratt Algorithm [KMP77]. It achieves linear worst case running time by precomputing information about matches and mismatches which is used to avoid repeated comparisons. Our implementation of the Knuth-Morris-Pratt Algorithm is based on $\mathcal{A}^+$-trees. Section 4.3 is devoted to the Karp-Rabin Algorithm [KR87] which applies hash techniques to obtain a speedup in the average case. A large part of chapter 4 is devoted to the Boyer-Moore Algorithm [BM77]. The basic idea of this algorithm is to align the pattern with a fragment of the input string and to look for a match by comparing the aligned characters from right to left. The information gathered in this comparison is used to determine a shift of the pattern to the

right. In sections 4.4.1 and 4.4.2, we consider two heuristics to determine such a shift. We show how to preprocess the pattern and the input alphabet in order to apply the heuristics efficiently. The main contribution of section 4.4.1 is a new and relatively simple linear time preprocessing method to be used in the Boyer-Moore Algorithm. This method utilizes the suffix tree of the pattern which has been augmented with some extra information. In sections 4.4.4 and 4.4.5, we consider Horspool's [Hor80] and Sunday's [Sun90] heuristics which allow for a simplified and faster preprocessing phase. Both heuristics lead to exact string searching algorithms with good average case behavior. In section 4.4.6, we show how to implement the Boyer-Moore Algorithm and its variations. Section 4.5 is devoted to the widely unknown exact string searching algorithm of Chang and Lawler [CL90]. We have included this algorithm because it combines the advantages of the Knuth-Morris-Pratt Algorithm and the Boyer-Moore Algorithm in an elegant way. This is accomplished by traversing the suffix tree of the pattern in two different manners. Section 4.6 summarizes the properties of the functional implementations given in chapter 4.

In chapter 5, we consider the multiple exact string searching problem which consists of finding exact occurrences of several patterns in an input string. We describe the suffix tree search algorithm which preprocesses the input string into a suffix tree in order to achieve optimal search time. Section 5.2 deals with the Aho-Corasick Algorithm [AC75] which preprocesses the patterns into an $\mathcal{A}^+$-tree in order to look for all patterns in the input string simultaneously. In section 5.3, we briefly describe other approaches for solving the multiple exact string searching problem.

Chapter 6 deals with the approximate string searching problem which consists of finding approximate occurrences of a pattern in an input string. We measure the approximation quality by the edit distance. Except for one,[4] all functional implementations achieve the same asymptotic efficiency as their imperative counterparts. In section 6.1, we describe Sellers' [Sel80] dynamic programming solution to the approximate string searching problem. Since Sellers' method is a variation of the Wagner-Fischer Algorithm, it can easily be implemented using the framework developed in section 3.9.5. This is also true for the "cutoff" variation of Sellers' method (which was suggested by Ukkonen [Ukk85b]). In section 6.2.1, we analyze the combinatorics of the dynamic programming table computed by Sellers' method. This was already done in [Ukk93a], but we present some simpler proofs and additional properties. We give a declarative description of Ukkonen's algorithm [Ukk93a] which traverses the suffix tree of the input string to solve the approximate string searching problem. Our description abstracts from a concrete data structure which makes it quite amenable for the correctness proof given in section 6.2.1. In section 6.2.2, we briefly discuss the implementation techniques suggested in [Ukk93a]. Section 6.2.3 presents a new online implementation technique for Ukkonen's algorithm. In section 6.2.4, we develop the ideas presented in [Ukk93a] one step further. We describe a new algorithm that incrementally computes a deterministic finite automaton for solving the approximate string searching problem. For the remainder of chapter 6, we restrict ourselves to the unit cost model. This leads to an instance of the approximate string searching problem widely known as the k-differences problem. After briefly recalling the combinatorics of the problem, we describe the most important solutions in detail beginning with Ukkonen's "cutoff" algorithm [Ukk85b]. Section 6.5 is devoted to Ukkonen's [Ukk85b] deterministic finite automaton which is obtained by precomputing each column possibly occurring in the dynamic programming table.

[4] The $\mathcal{O}(k \cdot n)$ worst case algorithm of [LV88].

We present a functional program that constructs the automaton incrementally. Section 6.6 deals with the Diagonal Transition Algorithm which was devised in [LV88]. Our declarative presentation of the algorithm contrasts the traditional imperative views. We present two variations of the algorithm. A brute force method which runs fast in practice, and a variation that achieves an improved worst case performance by preprocessing the pattern and the input string. The preprocessing methods of [CL94] and [UW93] are considered in section 6.6.3. A main contribution is a declarative description and detailed correctness proof of Chang and Lawler's linear time algorithm for computing matching statistics. Section 6.6.4 considers implementation issues. Unfortunately, the functional implementation of the preprocessing Diagonal Transition Algorithm does not achieve the improved worst case running time as it is possible for an imperative implementation. In section 6.7, we present Chang and Lampe's Column Partition Algorithm [CL92]. Our presentation contains some important details omitted in [CL92]. We derive an efficient functional implementation in a systematic way by program transformation. The algorithms presented in section 6.8 are due to Chang and Lawler [CL94]. The idea is to apply a filter to the input string to eliminate large portions that cannot contain an approximate match. By modifying a notation of Ehrenfeucht and Haussler [EH88] we can describe Chang and Lawler's filter techniques in a very concise and intuitive way. This clarifies the boundary cases and reveals some sources for improving the techniques. In section 6.8.6, we briefly describe two further approaches for solving the k-differences problem. Section 6.9 summarizes the properties of the functional implementations given in chapter 6.

Chapter 7 gives some performance results for the string searching algorithms we have implemented. In chapter 8, we recall the main findings of the thesis and outline some topics of future work.

Chapter 2

Functional Programming Concepts

Imperative programming languages such as FORTRAN, Pascal, C and Ada make essential use of assignments to variables as the basic construct, around which are built the control abstractions of sequencing, branching and looping, and so on. For a given input, an imperative program determines a sequence of instructions that transform a store (made up of cells) by updating the contents of the cells. The imperative programmer has to organize the computation sequencing into small steps and to care about the memory management, that is, the use and reuse of variables and the distribution of large values across many cells. On the one hand, this fine control allows to write programs which are hard to beat in terms of space and time. On the other hand, the necessity to provide the administrative details of sequencing and memory management leads to some well-known disadvantages:

- The productivity of the programmer is low.
- The programs are often error prone.
- It is problematic to port the programs to other machine architectures, especially highly parallel machines.
- It is difficult to treat the programs by formal proof methods.
- Parts of the programs are often not reusable.

A good example for the benefits of liberating the programmer from administrative details can even be found in most imperative languages with the concept of arithmetic expressions. By providing an appropriate arithmetic expression, the programmer only has to describe *what* is to be computed. The details of *how* it is computed are left to the compiler. Consider the following arithmetic expression (see [Rea89])

$$((i + j) * (\text{abs } (i - j) + 1)) \text{ div } 2$$

which describes the sum of the integers between two given integers i and j. It contains the application of several functions as well as referring to data values i, j, 1 and 2. Nothing is said about certain details, such as where the intermediate results are stored. We can also see the potential for several different orders of evaluation, including parallel evaluations of subexpression, such as $(i + j)$ and $(\text{abs } (i - j) + 1)$. Another important point is that the

occurrences of i and j stand for integer values rather than cells in which values may be placed.

One of the most important ideas of the functional programming paradigm is to *completely* liberate the programmer from sequencing and memory management details. This is achieved by using expressions as the uniform basis for programming. In particular, programming in a functional language consists of defining functions. The primary role of the computer is to evaluate expressions and to print the results. An expression which contains occurrences of the names of functions is evaluated by using the function definitions as reduction rules. Let us briefly recall the most important properties of functional programming.

Referential Transparency According to [Sto77], referential transparency means the following:

> "The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. Moreover, the value of an expression is, within certain limits, the same whenever it occurs."

Therefore, an expression in a functional language is free of side effects. It can be constructed, manipulated and reasoned about, like any other kind of mathematical expression, using more or less familiar algebraic laws. Moreover, since the evaluation of different subexpressions does not affect each other, the value of an expression is independent of the order in which it is evaluated. This principle is known as the confluence or Church-Rosser property. It allows the programmer to devise an algorithm without considering the control flow. Therefore, functional languages are very suitable for programming highly parallel computers (see [PJ89]).

Abstraction This is a very important property in programming since it allows for separation of concerns. Functional languages allow a high degree of abstraction. This is mainly achieved by considering functions as values. That is, functions can be passed as arguments to other functions and returned as results. This concept of higher order functions considerably simplifies the structuring, the modification, the composition, and the modularization of programs and leads to a higher degree of reusability.

Polymorphic Typing Like monomorphic typing which is used in Pascal, polymorphic typing eliminates many sources for errors. However, it is more flexible and considerably improves the reusability.

Notational Convenience The notation of functional languages obeys normal mathematical principles. It is very simple, concise, and powerful. Hence, for many problems, functional languages are an ideal tool to write down executable specifications.

For a more detailed discussion of these topics we refer the reader to the textbooks [Hen80, GHT84, Wik87, BW88, Rea89, Mac90, Hin92, Thi94].

Note that there are functional languages that are not referentially transparent, since they allow for impure features like assignments (for example, Standard ML [MTH90] and most dialects of LISP [McC60]). We refer to these as *impure* functional languages. In the thesis we only consider *pure*, that is, referentially transparent functional languages.

Functions which are always undefined with undefined arguments are called *strict* whereas functions which can give defined results even when arguments are undefined are called *non-strict*. Functional languages like Standard ML or Scheme [RC86] assume strict functions and use *eager evaluation*. This means that arguments to functions are fully evaluated before the function is applied. Functional languages like Miranda [Tur85], Lazy ML [AJ89], Haskell [FHPJW92], Gofer [Jon94], and Clean [PE95] support non-strict functions and are based on *lazy evaluation*. This means that the evaluation of function arguments is postponed until the value is known to be required.

The programs described in the thesis are written in a lazy functional language. This is for the following advantages of lazy evaluation:

- It is often the case that the most natural specification of a function is closer to being an efficient program for lazy evaluation than it is for eager evaluation [Rea89].
- Laziness allows to use infinite data structures in an elegant way. This provides a further mechanism to separate control and calculation and thus improves modularization and reusability. Consider streams, which, according to Abelson and Sussman [AS84], are a very powerful tool for structuring programs. In a lazy functional language, streams are nothing else but infinite lists, whereas in an eager functional language, it is necessary to introduce a special syntax for streams like it is done in Scheme.
- Whenever an expression has a defined value it can be computed by lazy evaluation. In this sense, lazy evaluation is fully consistent with the principle of referential transparency [Tur85] and programs may be written without any concern for control flow. This is not true for eager evaluation.

As the thesis should also be valuable to the programmers who prefer eager functional languages, we will always point out where in our programs the laziness is important. In most cases, standard techniques (see [Rea89]) can then be applied to derive variants of our functions that are efficient for eager evaluation.

We will use the lazy functional language Miranda as a notation for our programs. Miranda is a widespread language with a simple but mature implementation [Tur86, Tur89]. Because of its terse syntax, it is used in several textbooks (see [PJ87, BW88, PJL91, Hol91, Hin92, CMP94]) as well as in some papers (see [MLC91, NS93, HL93, GK94]). We restrict ourselves to those features of Miranda that are (with similar syntax and semantics) also present in the lazy functional languages Gofer, Haskell, and Clean. Hence, it is straightforward to translate our programs to these languages.

In the following section, we give an introduction to the basic concepts and notations of Miranda. Similar to Turner [Tur86, PJ87], this will be done in an informal way by several simple examples. However, our introduction is tailored to the specific needs of the application field. So we mostly take examples of simple string processing functions.

2.1 An Introduction to Miranda

A Miranda program is a collection of equations defining various functions and sets of values, called types. Expressions consist of functions and values and they themselves denote a value. The Miranda system evaluates expressions by using the function definitions as reduction rules. Let us give an example. To indicate a Miranda program, we use `typewriter font`. Miranda expressions that occur in running prose are printed in *emphasize font*, mostly surrounded with brackets for better readability.

```
list α ::= Empty | Cons α (list α)

length::(list α)→num
length Empty = 0
length (Cons a x) = 1 + length x
```

In the first line above, a type $(list\ \alpha)$ is introduced. It consists of the values *Empty* and $(Cons\ a_1\ (\cdots(Cons\ a_n\ Empty)\cdots))$, for all $n \geq 1$ where a_i is a value of type α, $1 \leq i \leq n$. The rest of the program defines a function *length* which takes a value of type $(list\ \alpha)$ as an argument and returns a numeric value of type *num*. The two equations defining *length* can be used as reduction rules to evaluate the expression $(length\ (Cons\ 0\ (Cons\ 1\ Empty)))$ to 2 as follows:

$$\begin{aligned} length\ (Cons\ 0\ (Cons\ 1\ Empty)) &\Rightarrow 1 + length\ (Cons\ 1\ Empty) \\ &\Rightarrow 1 + 1 + length\ Empty \\ &\Rightarrow 1 + 1 + 0 \\ &\Rightarrow 2 + 0 \\ &\Rightarrow 2 \end{aligned}$$

The sign $\Rightarrow$ should be read as "reduces in one step to". Note that for the last two reduction steps the predefined equations $1 + 1 = 2$ and $2 + 0 = 2$ are used. In the following, we shall be more specific about Miranda's type systems and the notation of functions.

2.1.1 Types

Miranda is polymorphic and strongly typed. This means the following:

- Sometimes in functions and data structures no particular property of the elements of a type is required. This situation can be handled by using polymorphic types which essentially describe families of types. Polymorphic types in Miranda contain universally quantified type variables $*$, $**$, $***$, and so on. To improve readability, we adopt the notation of [BW88] and use greek letters α, β, γ, and so on, for type variables.
- Any inconsistency in the type structure results in a type error at compile time. Thus, every compiled Miranda program is type-safe. That is, every expression has a unique most common type. This helps in reasoning about programs and permits to generate efficient code since at running time no type information is required.

We have already seen an example of polymorphism. The type (*list* α) is polymorphic, that is, it contains values over an arbitrary type α. Special instances of this type are (*list num*), (*list bool*) and (*list* (*list bool*)). The function *length* is polymorphic, since no particular property of the name a is required in the definition of *length*.

With the equation defining (*list* α), we have seen the uniform way of introducing user-defined types in Miranda. Additional examples are a type for weekdays and a type for binary trees with node labels of arbitrary type:

```
weekday ::= Mon | Tue | Wed | Thu | Fri | Sat | Sun
binarytree α ::= Nil | Binary α (binarytree α) (binarytree α)
```

These equations introduce constructors, that is, identifiers constructing types. In particular, the type *weekday* consists of 7 values, the constructors $Mon, \ldots, Sun$. The type (*binarytree* α) consists of infinitely many values which are built from the type constructors *Nil* and *Binary* and the constructors for the type α. An example is the value

```
Binary 7 (Binary 8 Nil Nil) (Binary 9 Nil Nil)
```

of type (*binarytree num*). Note that a user-defined constructor always begins with a capital letter.

Miranda has several predefined types. There are three primitive types *num*, *bool* and *char* containing numerical values, boolean values (*True* and *False*) and ASCII-characters. Moreover, we can make use of three predefined compound types, which are constructed according to the following rules:

- If α is a type, then $[\alpha]$ is a type. It contains the value $[\,]$. Moreover, if a is of type α and x is of type $[\alpha]$, then $(a : x)$ is of type $[\alpha]$. The values of type $[\alpha]$ are called *lists*, $[\,]$ is the constructor for the empty list, and (:) is the infix constructor for a non-empty list.

- Let $n = 0$ or $n \geq 2$. If α_1 to α_n are types, then $(\alpha_1, \ldots, \alpha_n)$ is a type. It contains the *n-tuples* of the form $(a_1, \ldots, a_n)$ where a_i is of type α_i, $1 \leq i \leq n$.

- If α and β are types, then $\alpha \to \beta$ is a type. It contains all functions with arguments of type α and results of type β. Note that $\to$ is right associative, that is, $\alpha \to \beta \to \gamma$ means $\alpha \to (\beta \to \gamma)$.

Lists can be written with square brackets and commas separating the elements. So $[1, 2, 3]$ is the shorthand notation for $(1 : (2 : (3 : [\,])))$. Note that the symbol () is overloaded. It denotes a type, and the only element of this type, the *nullary tuple*. There are no 1-tuples, since for an element a of type α the expression (a) is of type α by convention.

In principle, it is not necessary to have predefined types since for each predefined type we can introduce an isomorphic user-defined type. For instance, (*list* α) is isomorphic to the type $[\alpha]$, that is, there is a bijective function from (*list* α) to $[\alpha]$. *Empty* maps to $[\,]$ and (*Cons a x*) maps to $(a : x)$. The reason for having predefined types is that they allow for a more convenient notation and considerably improve the readability and efficiency of the programs.

There are several predefined functions on the predefined types. Consider, for instance, lists: $(\#l)$ returns the length of the list l, the infix function $(+\!\!+)$ appends two lists, and $(l!i)$ returns the i-th list element in l for $0 \leq i \leq \#l - 1$. Some of the predefined functions are explained in this chapter. For definitions of the other predefined functions appearing in the thesis we refer the reader to Appendix B.

Type declarations for functions are written with the symbol (::) as for the function *length* above. Since the Miranda system is able to deduce the type of a function from its defining equations, type declarations are not really necessary. However, for documentation purposes, they are very useful. Moreover, they support the system to produce helpful error messages. Therefore, we will always give a function with its type declaration. Sometimes it is inconvenient to spell out the type declarations in terms of the basic type names. For these cases, type synonyms allow for the introduction of new type names for an already existing type. As an example, consider type synonyms for dates:

```
month == num
day == num
date == (weekday,month,day)
```

2.1.2 Functions

In Miranda, functions are defined by equations. An equation defining a function, say f, can contain patterns, that is, expressions built from variables and constructors. To apply an equation defining f, the patterns have to match the actual parameters of f. Suppose f is a function on lists. Then the identifier y is a pattern that matches any list, the constructor [ ] is a pattern that matches the empty list, and the expression $(a : x)$ is a pattern that matches any non-empty list binding a to the first element and x to the remainder of the list. If a match succeeds, then the matching subexpression is replaced by the right hand side of the equation after substituting bound variables. If a match fails, the next equation defining f is tried. As an example consider the functions *hd* and *suffixes*.

```
hd::[α]→α
hd (a:x) = a

suffixes::[α]→[[α]]
suffixes [] = [[]]
suffixes (a:x) = (a:x):suffixes x
```

hd returns the first element of a non-empty list. Note that *hd* is undefined for an empty list. *suffixes* returns the list of all suffixes of a list ordered by descending length. In particular, the only suffix of the empty list is the empty list itself. For a non-empty list of the form $(a : x)$ we get $(a : x)$ itself as a suffix together with all suffixes of x.

An equation defining a function can have several alternative right hand sides distinguished by boolean expressions, called *guards*. The semantics specifies that the guards are tested in order, from top to bottom. The keyword *otherwise* may be used as the last guard, indicating that this is the case which applies when all the other tests fail. This is illustrated by the predefined function *merge* which merges two ordered list into a single ordered list.

```
merge::[α]→[α]→[α]
merge [] y = y
merge (a:x) [] = a:x
merge (a:x) (b:y) = a:merge x (b:y), if a<=b
                  = b:merge (a:x) y, otherwise
```

Note that a function application is simply written by juxtaposition of the function and the arguments. This reduces the number of brackets which have to be written in expressions. Since function application is left associative, an expression $(f\ a\ b)$ is parsed as $((f\ a)\ b)$, meaning that the result of applying f to a is a function which is then applied to b. So $(+3)$ is a function of type $(num \to num)$ that adds 3 to its argument. This device which applies to any function of two or more arguments is known as *currying* (after the logician H.B. Curry).

Local definitions can be introduced on the right hand side of a function definition, by the means of a *where* clause. This is illustrated by the function *split* which splits a list into a prefix of a given length and the remaining suffix. Note that the definition below contains the patterns 0 and $(i+1)$. The pattern 0 matches only itself and the pattern $(i+1)$ matches any positive integer j, binding i to $(j-1)$.

```
split::num→[α]→([α],[α])
split 0 x = ([],x)
split (i+1) [] = ([],[])
split (i+1) (a:x) = (a:y,z)
                    where (y,z) = split i x
```

Miranda allows *higher order functions*. That is, functions can be both passed as parameters and returned as results. The most important higher order function is the composition (.) which is written in infix notation. It is defined as follows:

```
(.)::(α→β)→(γ→α)→γ→β
(f.g) x = f (g x)
```

The use of higher order functions is an important feature of functional programming. It often leads to a very concise form of expressions. As an example, consider the predefined functions *map*, *foldl*, and *foldr* which implement common recursion schemata on lists:

```
map::(α→β)→[α]→[β]
map f [] = []
map f (a:x) = f a:map f x

foldl::(α→β→α)→α→[β]→α
foldl f a [] = a
foldl f a (b:x) = foldl f (f a b) x

foldr::(α→β→β)→β→[α]→β
foldr f a [] = a
foldr f a (b:x) = f b (foldr f a x)
```

map applies f to each element of a list, i.e., $map\, f\, [x_1, x_2, \ldots, x_n] = [f\, x_1, f\, x_2, \ldots, f\, x_n]$.

foldl folds a list from the left, i.e., $foldl\, f\, a\, [x_1, x_2, \ldots, x_n] = f\, (\cdots (f\, (f\, a\, x_1)\, x_2) \cdots)\, x_n$.

foldr folds a list from the right, i.e., $foldr\, f\, a\, [x_1, x_2, \ldots, x_n] = f\, x_1 (f\, x_2 (\cdots (f\, x_n\, a) \cdots))$.

A lot of useful functions can be obtained using *map*, *foldl*, and *foldr*. Consider, for instance, the predefined function *sum* that computes the sum of a list of numbers, and the function *or* that returns the disjunction of a list of boolean values.

```
sum::[num]→num
sum = foldl (+) 0

or::[bool]→bool
or = foldr (∨) False
```

The ( )-notation is Miranda syntax to convert an infix operator (for instance, $\vee$ or $+$) into an ordinary binary operator. Let us give some more examples of higher order functions.

```
member::[α]→α→bool
member x a = or (map (a=) x)

reverse::[α]→[α]
reverse = foldl revcons []
          where revcons x a = a:x

prefixes::[α]→[[α]]
prefixes = foldr extend [[]]
           where extend a ps = []:map (a:) ps
```

member checks if a value a is present in a list x. *reverse* applied to any finite list returns a list of the same elements in reverse order. *prefixes* returns the list of prefixes of a list ordered by increasing length. *member* and *reverse* are predefined functions. We will make use of several predefined higher order functions. This avoids explicit recursion and makes programs more concise and readable.

2.1.3 List Comprehensions

List comprehensions give a concise syntax for a rather general class of iterations over lists. A simple example is:

```
squares::num→[num]
squares n = [i * i | i←[1..n]]
```

(*squares* 100) returns the squares of the numbers from 1 to 100. The expression on the right hand side should be read as "list of i times i where i is drawn from $[1..n]$". The construct to the right of the bar is called a generator: The pattern introduced on the left hand side of the symbol $\leftarrow$ ranges over all elements on the right hand side that match the pattern. List comprehensions can contain several generators and also filters, that is, boolean expressions. This is illustrated by the functions *triads* and *select*. *triads* lists all Pythagorean triads in a given range, *select* returns all lists y such that $(a : y)$ occurs in x.

```
triads::num→[(num,num,num)]
triads n = [(i,j,k) | i←[1..n]; j←[i..n]; k←[j..n]; i^2 + j^2 = k^2]

select::α→[[α]]→[[α]]
select a x = [y | c:y←x; a = c]
```

Note that list comprehensions add no fundamental new power to a language. That is, any program containing list comprehensions can be translated to an equivalent program that does not contain them (see [PJ87]). However, due to their resemblance to set comprehensions in mathematics, list comprehensions greatly increase the ease with which one can write and read functional programs.

2.1.4 Non-Strict Functions and Infinite Data Structures

One consequence of lazy evaluation is that it allows to handle non-strict functions. Consider the function *cond*:

```
cond::bool→α→α→α
cond True x y = x
cond False x y = y
```

cond is non-strict which means that the expression ($cond\ False\ (1/0)\ 2$) evaluates to 2. *cond* exactly corresponds to the control structure provided by guarded equations. So the third equation defining the function *merge* (see section 2.1.2) can alternatively be written as follows:

```
merge (a:x) (b:y) = cond (a<=b) (a:merge x (b:y)) (b:merge (a:x) y)
```

Note that under a strict semantics the second and the third argument of *cond* are evaluated independently of the value of the first argument. Hence, in an eager language *cond* does not exactly correspond to guarded equations.

Lazy evaluation allows programming with unknowns. That is, we can simultaneously define and use an expression. As an example consider the function *scanl*, which applies *foldl* to every initial segment of a list. More precisely, the expression ($scanl\ f\ a\ [x_1, \ldots, x_n]$) returns the list $z = [z_0, z_1, \ldots, z_n]$ such that $z_0 = a$ and $z_{j+1} = f\ z_j\ x_{j+1}$ for each $j, 0 \leq j \leq n-1$.

```
scanl::(α→β→α)→α→[β]→[α]
scanl f a x = z
              where z = a:[f b c | (b,c)←zip2 z x]
```

The implementation of *scanl* relies on the fact that the list constructor (:) is non-strict. As a consequence, the expression z can be used on the right hand side of the equation defining z. Note that the first element of z is given. So *scanl* returns a welldefined result. *scanl* is equivalent to the predefined function *scan*. However, experimental results show that *scanl* is considerably faster than *scan*. For this reason we use *scanl* instead of *scan*.

An important virtue of lazy evaluation is that it allows to write down definitions of infinite data structures. Here are some examples of infinite lists of natural numbers and an infinite binary tree.

```
nats = [0..]
odds = [1,3..]
fibonaccis = f 0 1
             where f i j = i:f j (i+j)
infbinarytree = root
                where root = Binary 0 (Binary 1 root Nil) (Binary 2 root Nil)
```

(*repeat* a) returns an infinite list which contains only the element a. The function *enum* enumerates all, that is, infinitely many strings over a given alphabet.

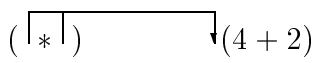
```
repeat::α→[α]
repeat a = repa where repa = a:repa

enum::[α]→[[α]]
enum alphabet = []:[a:w | w←words; a←alphabet]
                where words = enum alphabet
```

To obtain only the first 100 strings over the alphabet $\{a, b\}$, we can use a predefined function *take* and evaluate the expression (*take* 100 (*enum* $[a, b]$)). Similarly, the expression (*takewhile* (< 1000) *fibonaccis*) returns all Fibonacci numbers smaller than 1000. The laziness guarantees the efficiency of these definitions: The two lists are enumerated as demanded by the functions (*take* 100) and (*takewhile* (< 1000)). Note that in a language with strict functions the code for enumerating elements is intertwined with the code for selecting elements. That is, there is no separation between the logical separate phases. This leads to less flexible and reusable programs.

2.2 The Model of Computation

In order to evaluate an expression, the equations defining functions are used as reduction rules. Each reduction step replaces a reducible subexpression (*redex* for short) by an equivalent expression. We shall adopt the computation model of lazy evaluation which means that, in order to evaluate an expression, no subexpression is evaluated until its value is known to be required. Lazy evaluation uses a *leftmost outermost reduction* strategy. Every reduction step reduces an outermost redex, that is, a redex contained in no other redex. If there is more than one outermost redex, the leftmost is taken. To ensure that arguments are evaluated at most once, an expression is represented by a graph that indicates shared subexpressions. This is illustrated by an example from [BW88]: The graph



represents the expression $(4 + 2) * (4 + 2)$. If the function sq is defined by $(sq\ i = i * i)$ and the expression $(sq\ (4 + 2))$ is to be evaluated, then lazy evaluation leads to the following sequence of reduction steps:

$$
\begin{aligned}
sq\ (4+2) &\Rightarrow \quad (\boxed{*}\,)\qquad\qquad (4+2)\\
&\Rightarrow \quad (\boxed{*}\,)\qquad\qquad 6\\
&\Rightarrow \quad 36
\end{aligned}
$$

Hence, the expression (4+2) is evaluated only once instead of twice when ordinary outermost reduction is applied. With the representation of expressions by graphs, lazy evaluation never needs more reduction steps than eager evaluation.

The time and space requirement of a computation is measured in terms of outermost graph reduction steps. We say that an expression e is evaluated to the value v using n steps and $\mathcal{O}(m)$ space if the following holds:

- The sequence
$$e = g_0 \Rightarrow g_1 \Rightarrow g_2 \Rightarrow \cdots \Rightarrow g_n = v \tag{2.1}$$
of outermost graph reduction steps is of length n.
- The largest graph in the sequence (2.1) is of size $\mathcal{O}(m)$.

Note that the number of outermost graph reduction steps does not correlate reliably with actual execution time, because some kinds of reduction are much slower than others. Consider, for instance, the two equations defining functions f and g:

$$\begin{aligned} f\ (a : b : x)\ (i + 1)\ (j + 2, 0)\ &=\ 1000 \\ g\ a\ &=\ 7 \end{aligned}$$

Evaluating the expression $(f\ [1, 2]\ 5\ (3, 0))$ to 1000 and $(g\ 6)$ to 7 takes one outermost graph reduction step in both cases. However, due to the complicated left hand side an application of the first equation costs much more time.

The Miranda system provides a statistics facility which counts reduction steps. It is important to note that this statistics does not refer to outermost graph reduction steps. Instead, the work of an abstract machine is measured. More precisely, the Miranda system transforms each function into an expression that essentially consists of a small family of built-in functions, called SKI-combinators [Tur79]. These are interpreted by a very simple abstract machine. It is reasonable to assume that every reduction performed by this machine takes constant time. Hence, the actual execution time of a program is more or less proportional to the measured number of SKI-reductions. (For a SPARC 10/41 a constant rate of about 80.000 SKI-reductions per CPU-second was measured [BG93].) Moreover, every graph reduction step leads to a certain number of SKI-reductions which is constant for each left hand side of a function definition and independent of the input size. Thus, the asymptotic efficiency, in terms of outermost graph reduction steps, is not affected and can be validated with the statistics provided by the Miranda system. This will be done in Chapter 7.

Besides the SKI-reductions, the Miranda system also reports the number of cells claimed during an evaluation. However, it is unclear if this number can be used to validate the analytically determined space consumption of an evaluation.

2.3 Monads

Purely functional programming languages allow many algorithms to be expressed very concisely, but there are a few algorithms in which in-place updatable state seems to play a crucial role. Among these are algorithms based on the use of incrementally modified hash tables

where lookups are interleaved with the insertion of new items. Similarly, the union/find algorithm relies for its efficiency on the set representations being simplified each time the structure is examined. Likewise, many graph algorithms require a dynamically changing graph structure in which sharing is explicit, so that changes are visible non-locally.[1] For these algorithms, purely functional languages which lack updatable state, appear to be inherently inefficient (see [PMN88]).

Recent advances in theoretical computer science, notably in the areas of type theory and category theory, have suggested new approaches that allow to integrate updatable state into purely functional languages. For recent results see [Hud93]. One of the most important approaches in this field are *monads*. In several papers [Wad90, Wad92b, Wad92a], Wadler has shown that the essence of impure features can be captured by the use of monads in a purely functional programming language.

In this section, we review Wadler's idea. In particular, we give a short introduction to monads in functional programming and demonstrate that the monadic programming style is a helpful structuring technique to encapsulate state based computations. We especially show how one could integrate arrays with in-place updates in Miranda, without destroying the referential transparency.

Let M be a type identifier, that is, an identifier used to denote a type. The basic idea in converting a program to monadic style is this: a function of type $(\alpha \to \beta)$ is converted to a function of type $(\alpha \to M\ \beta)$. Thus, a function in monadic style accepts an argument of type α and returns a result of type β with a possible additional effect captured by M. As shown in [Wad92a], this effect may be to act on state, generate output or raise exception.

There are two basic functions *unit* and *bind* manipulating monadic values:

- *unit* is of type $(\alpha \to M\ \alpha)$. It takes a value of type α into its corresponding monadic value.
- *bind* is of type $(M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta)$. Informally *bind* gets us around the monad. *bind* is usually written in infix notation. In Miranda, this is expressed by prefixing *bind* with the symbol \$. The infix notation is used to express that the arguments are sequentially evaluated: *bind* has a monadic value of type $(M\ \alpha)$ as its first argument. It extracts a value of type α and applies a function to it, yielding a monadic value of type $(M\ \beta)$.

According to [Wad92b], a monad is defined as follows:

Definition 2.3.1 A *monad* is a triple $(M, unit, bind)$ consisting of a type identifier M and functions:

$$\begin{aligned} unit &:: \alpha \to M\ \alpha \\ bind &:: M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta \end{aligned}$$

such that for all $(a :: \alpha)$, $(m :: M\ \alpha)$, $(f :: \alpha \to M\ \beta)$ and $(g :: \beta \to M\ \gamma)$ the following laws hold:

$$(unit\ a)\ \$bind\ f = f\ a$$

[1] Examples of such graph algorithms are the linear time suffix tree constructions of McCreight [McC76] and of Ukkonen [Ukk93b] described in chapter 3.

$$m\ \$bind\ unit = m$$
$$(m\ \$bind\ f)\ \$bind\ g = m\ \$bind\ h$$

where $h\ a = (f\ a)\ \$bind\ g$. □

According to the three laws above, *bind* is sort of associative, and has *unit* as a left and right identity.

Example 2.3.2 [Wad92b] The trivial monad $(trivial, unit, bind)$ is defined by:

```
trivial α == α

unit::α→trivial α
unit a = a

bind::(trivial α)→(α→trivial β)→trivial β
m $bind f = f m
```

2.3.1 State Transformers

Imperative programming languages operate by assigning to a state. This is also possible in impure functional languages. In purely functional languages, assignment may be simulated by passing around a value representing the current state. We can express this by the monad $(statetrans\ \alpha\ \beta)$ of state transformers where α is the type of the states and β the type of the value returned by a state transformation.

```
statetrans α β == α→(β,α)

unit::β→statetrans α β
unit value state = (value,state)

bind::(statetrans α β)→(β→statetrans α γ)→statetrans α γ
(st $bind f) state = f value istate
                     where (value,istate) = st state
```

For instance, to mimic a counter, we can instantiate α by *num*. This yields an appropriate state transformer:

```
countertrans β == statetrans num β
```

A state transformer takes an initial state and returns a value paired with the new state. The function *unit* returns the given value and propagates the state unchanged. The higher order function *bind* takes the state transformer $(st :: statetrans\ \alpha\ \beta)$ and a function $(f :: \beta \rightarrow statetrans\ \alpha\ \gamma)$ and passes the initial state to *st*. This yields a *value* paired with an intermediate state named *istate*. $(f\ value)$ is a state transformer of type $(statetrans\ \alpha\ \gamma)$ which is applied to *istate*. This yields the result paired with the final state.

It is straightforward to verify for *statetrans* the laws in Definition 2.3.1 (see [Wad92b]). Thus, the triple $(statetrans, unit, bind)$ is indeed a monad. We now consider a special form of state transformers where the state is an array.

2.3.2 Array Transformers

Arrays play a central role in computing because they closely match current architectures. Programs are littered with array lookups such as $a[i]$ and array updates such as $a[i] := v$. These operations are popular because an array lookup is implemented by a single indexed fetch instruction, and an array update by a single indexed store. It is relatively easy to add arrays to a purely functional language if one restricts to provide only efficient array creation and lookup operations. How to incorporate an efficient array update operation, without destroying the referential transparency, is a question with a long history. Monads provide a new answer to this old question.

Let $(array\ \alpha)$ be the type of arrays with non-negative indices containing values of type α. The operations on this type are as follows:

```
makearray::num→[α]→array α
lookup::num→(array α)→α
update::num→α→(array α)→array α
```

Let $vlist = [v_0, v_1, \ldots, v_{n-1}]$ be a list of type $[\alpha]$.

- The call $(makearray\ n\ vlist)$ returns an array a with the indices $0, \ldots, n-1$ such that $a[i] = v_i$ for all $i, 0 \leq i \leq n-1$.
- The call $(lookup\ i\ a)$ returns the value in array a at index i.
- The call $(update\ i\ v\ a)$ returns an array where the entry i has value v and the remainder is identical to a.

The behavior of these operations satisfies the following laws:

$$lookup\ i\ (makearray\ n\ vlist) = v_i$$

$$lookup\ i\ (update\ j\ v\ a) = \begin{cases} v, & \text{if } i = j \\ lookup\ i\ a, & \text{otherwise} \end{cases}$$

makearray slightly generalizes the array creation operation *newarray* presented in [Wad90, Wad92a]. While *newarray* creates an array in which each entry is initialized with the *same* value, *makearray* performs the initializations according to a list of possibly different values. The efficient way to implement the *update*-operation is to overwrite the specified entry of the array. However, in a purely functional language this is only safe if there are no other references to the array when the *update*-operation is performed. An array satisfying this property is called *single threaded*, following [Sch85].

Example 2.3.3 [Wad92a] Consider the function *bad* defined as follows:

```
bad::num→α→α→array α→(array α,array α)
bad i v w a = (update i v a,update i w a)
```

Obviously, there are two references to the array a after applying the above equation. Therefore, the array a is not single threaded, and it is not safe to implement the update-operation by overwriting the entry $a[i]$. □

This example shows that an efficient update operation in a purely functional language requires that the arrays are single threaded. Monads provide a method to guarantee the single threadedness. The idea is to encapsulate the arrays into the monad *arraytrans* of array transformers. This monad is obtained from the monad of state transformers with the state taken to be an array:

```
arraytrans α β == statetrans (array α) β
```

Three new functions are added, corresponding to array creation, array lookup, and array update.

```
block::num→[α]→(arraytrans α β)→β
block n vlist at = b
                   where (b,a) = at (makearray n vlist)

fetch::num→arraytrans α α
fetch i a = (lookup i a,a)

assign::num→α→arraytrans α ()
assign i v a = ((),update i v a)
```

- Suppose *vlist* is as above. The call (*block n vlist at*) creates a new array a of size n such that $a[i]$ is initialized with v_i for all $i, 0 \leq i \leq n-1$. Then the array transformer *at* is applied to this array yielding a value b and an array a. Finally, the array is deallocated and b is returned.
- The call (*fetch i a*) returns the value at index i and the unchanged array.
- The call (*assign i v a*) returns the nullary tuple (), and updates the array so that index i contains value v. The typing of *assign* makes clear that the value returned is not of interest. In an impure language, one has an analogous function with result type (), indicating that the purpose of the function lies in its side effect.

A little thought shows that these operations are indeed single threaded. *fetch* is the only operation that could duplicate the array, thereby violating the single threaded property. To cope with this problem, *fetch* is implemented as follows (see [Wad92a, PJW91]): first the entry i in the array is fetched, and then the pair consisting of this value and the reference to the array is returned.

arraytrans is made into an abstract data type supporting the five functions *unit*, *bind*, *block*, *fetch*, and *assign*. *unit* and *bind* are inherited from the monad of state transformers. Each function manipulating an array must now be based on the data type *arraytrans*. This means that the type system of Miranda rejects each function outside the abstract data type that has an explicit array argument. In particular, the definition of the function *bad* (see Example 2.3.3) causes a type error. The type system totally guarantees that the arrays are single threaded, and hence updates can be implemented by overwriting without destroying the referential transparency. As noted in [LPJ94], such a reliance on the type system also occurs in other contexts. For instance, the implementation of the addition makes no attempt

to check that its arguments are indeed integers. In the same way, the array transformers make no attempt to ensure that there is only one reference to the array which is to be updated. This property is simply guaranteed by the type system.

Note that the operation *block* plays a central role in the abstract data type *arraytrans*. It is the only one that does not have the type identifier *arraytrans* in its result type. Without *block* there would be *no* way to write a program using *arraytrans* that did *not* have *arraytrans* in its output type.

2.4 Extensions to Miranda

Let us call a purely functional implementation of an algorithm *optimal* if it has the same asymptotic space and time complexity as an implementation of the algorithm in an imperative language. For most of the algorithms considered in the thesis we present an optimal functional implementation using only the basic language features of Miranda as they were described in section 2.1. However, in some cases the Miranda system would have to be slightly extended in order to achieve an optimal purely functional implementation. Since we are not able to incorporate these extensions in the Miranda system, we assume that they are given by some abstract data types. This clearly separates the extensions from the basic language features of Miranda and makes it easy to translate the programs to other purely functional languages which provide the extensions.

We need the following extensions to Miranda:

Extension 1 consists of a data type $(array\ \alpha)$ with the functions *makearray* and *lookup* as specified in section 2.3.2. We assume that a call to *lookup* takes constant time, and that *makearray* creates and initializes an array of size n in $\mathcal{O}(n)$ steps. See the index of the thesis for an account of the pages where extension 1 is used.

Extension 2 is like extension 1 but with an additional function *update* as specified in section 2.3.2. We assume that this function takes constant time. Extension 2 is mainly used for implementing linear time suffix tree algorithms (see section 3.6.3). Since these are relatively complicated, we use monadic programming style to guarantee single threadedness. Another application for extension 2 is the implementation of the preprocessing phase of Horspool's and of Sunday's exact string searching algorithms (see section 4.4.6). Since the preprocessing phase is quite simple, we do not use monadic programming style in this case. Instead, we directly apply the functions *makearray*, *lookup*, and *update*. The single threadedness is immediately clear.

To test the correctness of our implementations, we have realized the abstract data types in a straightforward way. Since the realizations are rather inefficient, we do not present them in the thesis. For the measurements of the running time one has to resort to a purely functional language that supports the extensions (for instance Haskell or Clean). For the case of linear time suffix tree algorithms this is done in a forthcoming master's thesis of Krause [Kra95].

Chapter 3

The String Processing Machinery

Suffix trees represent structured information about all subwords of a string. Hence, it is not surprising that they are an ubiquitous data structure with a myriad of applications to string processing problems, like repeat detection, multiple string searching, data compression, and approximate string searching. Sometimes there are better methods than those based on suffix trees. However, as pointed out by Apostolico [Apo85], there is no other form of index structure that seems to outperform suffix trees in versatility and elegance.

The algorithm of Wagner and Fischer [WF74][1] uses the technique of dynamic programming to compute the edit distance and the optimal alignments of two strings. Over the last two decades, this algorithm and its variations have become the most established tool for string comparisons and string searching. A variety of applications in computer science, molecular biology, speech processing, and coding theory is described by Kruskal and Sankoff [KS83].

Note that some authors have emphasized the importance of suffix trees and of the Wagner-Fischer Algorithm for string processing (see [Apo85, Mye91].) However, a systematic treatment of both concepts was (if at all) restricted to an abstract level. The main goal of this chapter is to establish a powerful string processing machinery by unifying both concepts on the implementation level. In particular, we show that a lazy functional language provides an ideal tool to handle various flavors of suffix trees, and many variations of the Wagner-Fischer Algorithm in a compact, flexible, and reusable framework. This leads to a collection of functions that serve as the building blocks of a system solving a variety of string searching and comparison problems. Note that in this chapter we focus on the main idea of the two concepts, as well as their functional implementation. The different problems solved by suffix trees and the Wagner-Fischer Algorithm are considered in detail in later chapters. We begin the description of our string processing machinery with some basic definitions and notations.

[1] According to [KS83], the basic idea underlying the Wagner-Fischer Algorithm was independently discovered and published by several other authors. For instance, Needleman and Wunsch [NW70] describe a variation which is of great importance in molecular biology applications.

3.1 Basic Definitions and Notations

Let S be a set. $|S|$ denotes the number of elements in S and $\mathcal{P}(S)$ refers to the set of subsets of S.

$\mathbb{N}_0$ denotes the set of positive integers including 0. $\mathbb{R}_0^+$ denotes the set of positive reals including 0. The symbols $h, i, j, k, l, m, n, q, r$ refer to integers if not stated otherwise. $|i|$ is the absolute value of i and $i \cdot j$ denotes the product of i and j.

Let M be a set and $f : M \times M \to \mathbb{R}_0^+$ be a function. f is a *metric* on M if for all $x, y, z \in M$ the following properties hold:

Zero Property	$f(x,y) = 0 \iff x = y$
Symmetry	$f(x,y) = f(y,x)$
Triangle Inequality	$f(x,y) \leq f(x,z) + f(z,y).$

Let $\mathcal{A}$ be a finite set, the *alphabet*. The elements of $\mathcal{A}$ are *characters*. Strings are written by juxtaposition of characters. In particular, ε denotes the *empty string*. The set $\mathcal{A}^*$ of *strings over* $\mathcal{A}$ is defined by

$$\mathcal{A}^* = \bigcup_{i \geq 0} \mathcal{A}^i$$

where $\mathcal{A}^0 = \{\varepsilon\}$ and $\mathcal{A}^{i+1} = \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^i\}$. $\mathcal{A}^+$ denotes $\mathcal{A}^* \setminus \{\varepsilon\}$. The symbols a, b, c, d refer to characters and $p, s, t, u, v, w, x, y, z$ to strings, unless stated otherwise.

The *length* of a string s, denoted by $|s|$, is the number of characters in s. We make no distinction between a character and a string of length one. If $s = uvw$ for some (possibly empty) strings u, v and w, then

- u is a *prefix* of s,
- v is a *subword* of s, and
- w is a *suffix* of s.

If u is a prefix of s, we write $u \sqsubset s$. If w is a suffix of s, we write $s \int w$.[2] A prefix or suffix of s is *proper* if it is different from s. A suffix of s is *nested* if it occurs more than once in s. A set S of strings is *prefix-closed* if $u \in S$ whenever $ua \in S$. A set S of strings is *suffix-closed* if $u \in S$ whenever $au \in S$. A subword v of s is *right-branching* if there are different characters a and b such that va and vb are subwords of s. Let $q > 0$. A *q-gram* of s is a subword of s of length q. The term q-gram is adopted from [Ukk92a]. Note that q-grams are sometimes called q-tuples (see [CM94, PW95]).

s_i is the i-th character of s. That is, if $|s| = n$, then $s = s_1 \ldots s_n$ where $s_i \in \mathcal{A}$. $s_n \ldots s_1$, denoted by s^{-1}, is the *reverse* of $s = s_1 \ldots s_n$. If $i \leq j$, then $s_i \ldots s_j$ is the subword of s beginning with the i-th character and ending with the j-th character. If $i > j$, then $s_i \ldots s_j$ is the empty string. A string w begins at position i and ends at position j in s if $s_i \ldots s_j = w$.

[2] Note that some authors write $w \sqsupset s$ when w is a suffix of s (for example, [CLR90]). We do not use this notation since it suggests that $w \sqsubset s$ implies $s \sqsupset w$, which is obviously not the case. Moreover, we found it convenient not to have related symbols for denoting the prefix and the suffix relation.

3.2 $\mathcal{A}^+$-Trees

In this section, we present the concept of $\mathcal{A}^+$-trees which occurs in many variants throughout the thesis. $\mathcal{A}^+$-trees especially serve as the "raw material" for suffix trees. They are also used to implement the Knuth-Morris-Pratt Algorithm and the Aho-Corasick Algorithm. Note that a large part of the standard terminology for suffix trees is generalized to $\mathcal{A}^+$-trees. This is necessary since some suffix tree constructions (see [McC76, Ukk93a]) apply standard operations to intermediate trees that are *not* suffix trees.

Definition 3.2.1 An $\mathcal{A}^+$-*tree* T is a rooted tree with edge labels from $\mathcal{A}^+$. For each $a \in \mathcal{A}$, every node k in T has at most one a-edge $k \xrightarrow{aw} k'$. □

Let T be an $\mathcal{A}^+$-tree. $edges(T)$ denotes the set of edges and $nodes(T)$ the set of nodes in T. $leaves(T)$ refers to the set of leaves and $inner(T)$ to the set of inner nodes in T. Hence, we have $nodes(T) = inner(T) \cup leaves(T)$. If T is the empty tree (the *root* with no edges), then the *root* is the only inner node. An edge leading to a leaf is a *leaf edge*.

The *size* of T, denoted by $|T|$, is the number of nodes in T. Let S be a set. An S-*annotation* of T is a function $\varphi : nodes(T) \to S$. Two $\mathcal{A}^+$-trees are *isomorphic* if they can be obtained from each other by renaming their nodes. More precisely, T and T' are isomorphic if there is a bijective function $\zeta : nodes(T) \to nodes(T')$ such that $k \xrightarrow{u} k' \in edges(T)$ if and only if $\zeta(k) \xrightarrow{u} \zeta(k') \in edges(T')$.

$path(k)$ denotes the concatenation of the edge labels on the path from the root of T to the node k. Due to the requirement of unique a-edges at each node of T, paths are also unique. Therefore, we denote k by $\overline{w}$ if and only if $path(k) = w$. The node $\overline{\varepsilon}$ is usually denoted by *root*. Let $\overline{w} \in nodes(T)$. $|w|$ is the *depth* of $\overline{w}$ and $T_{\overline{w}}$ is the *subtree* of T at node $\overline{w}$.

Definition 3.2.2 A string w *occurs* in T if T contains a node $\overline{wu}$, for some string u. $words(T)$ denotes the set of strings occurring in T. □

$\mathcal{A}^+$-trees are classified according to their degree of compression. We are interested in two special classes: atomic and compact $\mathcal{A}^+$-trees.

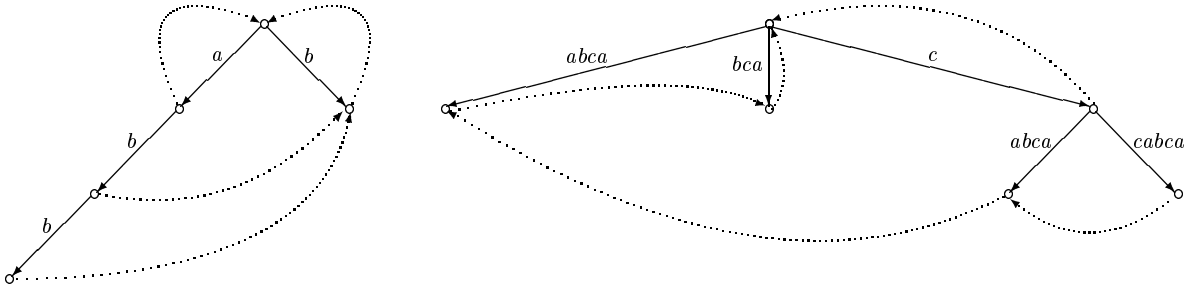
Definition 3.2.3 T is *atomic* if every edge in T is marked by a single character. T is *compact* if every node in T is either the *root*, a leaf, or a branching node. □

Atomic $\mathcal{A}^+$-trees are also known under the name trie [AHU82]. An atomic $\mathcal{A}^+$-tree is uniquely determined by the set of words occurring in it. The same holds for a compact $\mathcal{A}^+$-tree. We use the adjective compact since among all $\mathcal{A}^+$-trees with the same set of words occurring in them, the compact $\mathcal{A}^+$-tree is the smallest one.

3.2.1 Suffix Links

For some constructions and many applications it is convenient to augment $\mathcal{A}^+$-Trees with auxiliary edges connecting nodes that seem to be quite unrelated in the tree structure:

Figure 3.1: An Atomic and a Compact $\mathcal{A}^+$-tree with Suffix Links



Definition 3.2.4 Let $\overline{aw} \in nodes(T)$. Suppose v is the longest suffix of w such that $\overline{v}$ is a node in T. The unlabeled edge $\overline{aw} \longrightarrow \overline{v}$ is the *suffix link* for $\overline{aw}$. If $w = v$, then the suffix link for $\overline{aw}$ is *atomic*. The set of suffix links for all nodes in T (except for the *root*) is denoted by $links(T)$. In some cases we only need the *inner suffix links*, that is, the suffix links for the inner nodes of T (except for the *root*). We denote them by $innerlinks(T)$. □

Suffix links are usually defined as edges from a node $\overline{aw}$ to a node $\overline{w}$ (cf. [McC76, Ukk93b]). For general $\mathcal{A}^+$-trees this would, of course, lead to undefined suffix links since $\overline{w}$ may not exist. To cope with this problem, we follow [GK95b] and generalize the notion. Since $\overline{\varepsilon}$ is a node in T and ε is a suffix of w, each node $\overline{aw}$ in T has a suffix link. In the terminology of [AC75] the suffix links in an atomic $\mathcal{A}^+$ tree are called *failure transitions*. Some authors also define a suffix link $root \longrightarrow root$ (cf. [McC76, Ukk93b]). We found that this obscures the algorithms using suffix links.

Example 3.2.5 Figure 3.1 shows an atomic and a compact $\mathcal{A}^+$-tree with suffix links represented by unlabeled dotted edges. In the atomic $\mathcal{A}^+$-tree (on the left) the suffix links for the nodes $\overline{a}$, $\overline{b}$ and $\overline{ab}$ are atomic. In the compact $\mathcal{A}^+$-tree (on the right) the suffix links for the nodes $\overline{abca}$, $\overline{c}$, $\overline{cabca}$ and $\overline{ccabca}$ are atomic. All other suffix links are not atomic. □

3.2.2 Locations

If T is an atomic $\mathcal{A}^+$-tree, then there is a bijective function $\psi : words(T) \to nodes(T)$, defined by $\psi(s) = \overline{s}$. If T is a compact $\mathcal{A}^+$-tree, then such a bijective function does not exist in general since a string $s \in words(T)$ possibly "ends inside an edge". However, every string occurring in T can uniquely be described in terms of the nodes and edges of T:

Definition 3.2.6 Let T be a compact $\mathcal{A}^+$-tree and $s \in words(T)$. The *location* of s in T, denoted by $loc_T(s)$ is defined as follows:

- If $\overline{s} \in inner(T)$, then $loc_T(s) = \overline{s}$.
- If $\overline{s} \in leaves(T)$, then there is a leaf edge $\overline{u} \xrightarrow{v} \overline{s}$ in T and $loc_T(s) = (\overline{u}, v, \varepsilon, \overline{s})$.
- If there is no node $\overline{s}$ in T, then there is an edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ in T such that $s = uv$, $v \neq \varepsilon$, $w \neq \varepsilon$ and $loc_T(s) = (\overline{u}, v, w, \overline{uvw})$.

$locations(T) = \{loc_T(s) \mid s \in words(T)\}$ is the *set of locations* in T. If a location is a node, we call it *node location*, otherwise *edge location*. Sometimes we identify a node location with the corresponding node. □

By convention the *root* is the location of ε in the empty $\mathcal{A}^+$-tree. As locations allow a unified representation of the strings occurring in a compact $\mathcal{A}^+$-tree, they are a central data structure in many algorithms described in the thesis.

The location of a string corresponding to a leaf is not the leaf itself. Instead, it is defined in terms of the edge leading to the leaf. At the end of section 3.3.3 we give the reason for this. Note that an edge-location $(\overline{u}, v, w, \overline{uvw})$ corresponds to a leaf if and only if $w = \varepsilon$. In an edge-location $(\overline{u}, v, w, \overline{uvw})$ the string w and the node $\overline{uvw}$ are redundant. Both can uniquely be determined from $\overline{u}$ and v provided the $\mathcal{A}^+$-tree is given. If we omit w and $\overline{uvw}$, we get canonical reference pairs as they were introduced in [Ukk93b]. In some applications the third and fourth component of an edge-location is not used at all. However, in several cases, having them at hand considerably simplifies the algorithmic description.

For convenience we introduce some additional notions related to locations.

Definition 3.2.7 Let T be a compact $\mathcal{A}^+$-tree. Suppose $s \in words(T)$.

1. $|loc_T(s)| = |s|$ is the *depth* of the location $loc_T(s)$.
2. Let v be the shortest string such that $\overline{sv} \in nodes(T)$. Then $\overline{sv}$ is denoted by $ceiling(loc_T(s))$.
3. For all $a \in \mathcal{A}$ we define: $occurs(loc_T(s), a) \iff sa$ occurs in T.
4. $getloc(loc_T(s), w)$ denotes $loc_T(sw)$ for all $sw \in words(T)$. □

In the terminology of McCreight [McC76] a node location $\overline{s}$ is called *locus* of s. Moreover, $ceiling(loc_T(s))$ is the *extended locus* of s. The term *ceiling* is adopted from [CL94]. The function *getloc* is the analogue to the function *canonize* in [Ukk93b] which was defined for reference pairs.

Example 3.2.8 Let T be the compact $\mathcal{A}^+$-tree shown in Figure 3.1. Then, for instance,

$$
\begin{array}{lcllcl}
loc_T(\varepsilon) & = & root & ceiling(loc_T(\varepsilon))) & = & root \\
loc_T(a) & = & (root, a, bca, \overline{abca}) & ceiling(loc_T(a)) & = & \overline{abca} \\
loc_T(abca) & = & (root, abca, \varepsilon, \overline{abca}) & ceiling(loc_T(abca)) & = & \overline{abca} \\
loc_T(c) & = & \overline{c} & ceiling(loc_T(c)) & = & \overline{c} \\
loc_T(cab) & = & (\overline{c}, ab, ca, \overline{cabca}) & ceiling(loc_T(cab)) & = & \overline{cabca}
\end{array}
$$

□

Following a path is the most important operation on suffix trees. In some applications we have to perform this operation starting from an arbitrary location, going down as far as possible. We describe this by a function *scanprefix*.

Definition 3.2.9 Let T be a compact $\mathcal{A}^+$-tree. For each $s \in words(T)$ and each string w the function $scanprefix : locations(T) \times \mathcal{A}^* \to locations(T) \times \mathcal{A}^*$ is specified as follows: $scanprefix(loc_T(s), w) = (loc_T(su), v)$, where $uv = w$ and u is the longest prefix of w such that $su \in words(T)$. □

Example 3.2.10 Let T be the compact $\mathcal{A}^+$-tree as shown in Figure 3.1. Then, for instance, $scanprefix(loc_T(cab), cd) = (loc_T(cabc), d)$ and $scanprefix(loc_T(bca), d) = (loc_T(bca), d)$. □

For some suffix tree constructions to be described in section 3.5 we need to insert strings into an $\mathcal{A}^+$-tree. Therefore, we introduce the following operation:

Definition 3.2.11 Let T be a compact $\mathcal{A}^+$-tree. Suppose a is a character and s is a string such that $s \in words(T)$, $sa \notin words(T)$ and s does not correspond to a leaf in T. For each string y the expression $T[loc_T(s) \leftarrow ay]$ denotes the pair $(T', \overline{z})$ which is specified as follows:

- If $loc_T(s) = \overline{s}$, then T' is obtained from T by adding a leaf edge $\overline{s} \xrightarrow{ay} \overline{say}$. Moreover, $\overline{z} = \perp$ which should be read as "$\overline{z}$ is undefined".
- If $loc_T(s) = (\overline{u}, v, w, \overline{uvw})$, then T' is obtained from T by splitting the edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ into $\overline{u} \xrightarrow{v} \overline{s} \xrightarrow{w} \overline{uvw}$, and adding a new leaf edge $\overline{s} \xrightarrow{ay} \overline{say}$. Moreover, $\overline{z} = \overline{s}$, that is, $\overline{z}$ is the new inner node created by the splitting.

In some contexts we do not need to identify $\overline{z}$ and omit it so that $T[loc_T(s) \leftarrow ay]$ denotes T' alone. □

The insert operation subsumes the function *create-state* in [Ukk93b] which was introduced for splitting some edge. Note that our notation [$\leftarrow$] has nothing to do with list comprehensions. It stems from the field of term rewriting [HO80] where it has similar semantics.

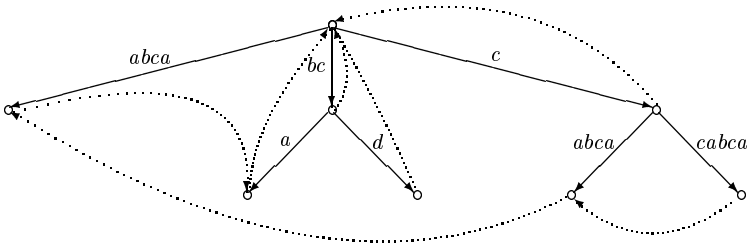
Lemma 3.2.12 Let T be a compact $\mathcal{A}^+$-tree. Suppose a is a character and s is a string such that $s \in words(T)$, $sa \notin words(T)$ and s does not correspond to a leaf in T. For each string y the following is true: $T' = T[loc_T(s) \leftarrow ay]$ is a compact $\mathcal{A}^+$-tree, $\overline{s}$ is a node in T' and $words(T') = words(T) \cup \{sav \mid v \sqsubset y\}$.

Proof Routine. □

Example 3.2.13 Let T be the compact $\mathcal{A}^+$-tree of Figure 3.1. Then the compact $\mathcal{A}^+$-tree $T[loc_T(bc) \leftarrow d]$ is shown in Figure 3.2. □

For some constructions and many applications of compact $\mathcal{A}^+$-trees it is very important to have an efficient access from $loc_T(cy)$ to $loc_T(y)$. This access is provided by a function *linkloc*, that uses the suffix links of the inner nodes as a "shortcut".

Figure 3.2: The Result of an Insert-Operation



Definition 3.2.14 Let T be a compact $\mathcal{A}^+$-tree. We define the function $linkloc : locations(T) \setminus \{root\} \to locations(T)$ as follows:

$$linkloc(\overline{s}) = \overline{z}$$
where $\overline{s} \longrightarrow \overline{z}$ is the suffix link for $\overline{s}$

$$linkloc(\overline{u}, av, w, \overline{uavw}) = \begin{cases} loc_T(v), & \text{if } \overline{u} = root \\ getloc(\overline{z}, av), & \text{otherwise} \end{cases}$$
where $\overline{u} \longrightarrow \overline{z}$ is the suffix link for $\overline{u}$. □

Note that *linkloc* never uses the suffix link of a leaf.

Lemma 3.2.15 Let T be a compact $\mathcal{A}^+$-tree such that all inner suffix links of T are atomic. Suppose that cy and y occur in T. Then $linkloc(loc_T(cy)) = loc_T(y)$.

Proof

1. Suppose $\overline{cy} \in inner(T)$. Then $\overline{cy} \longrightarrow \overline{y} \in innerlinks(T)$ and $linkloc(loc_T(cy)) = linkloc(\overline{cy}) = \overline{y} = loc_T(y)$.
2. Suppose $\overline{cy} \in leaves(T)$ or there is no node $\overline{cy}$ in T. Then $loc_T(cy) = (\overline{u}, av, w, \overline{uavw})$ and $cy = uav$. Notice that $\overline{u}$ is an inner node.
   - If $\overline{u} = root$, then $av = cy$ and therefore $v = y$. Hence, we get $linkloc(loc_T(cy)) = linkloc((\overline{u}, av, w, \overline{uavw})) = loc_T(v) = loc_T(y)$.
   - If $\overline{u} \neq root$, then there is an atomic suffix link $\overline{u} \longrightarrow \overline{z}$ such that $u = bz$ for some $b \in \mathcal{A}$. Therefore, $bzav = uav = cy$ which implies $zav = y$. Hence, we get $linkloc(loc_T(cy)) = linkloc((\overline{u}, av, w, \overline{uavw})) = getloc(\overline{z}, av) = loc_T(zav) = loc_T(y)$. □

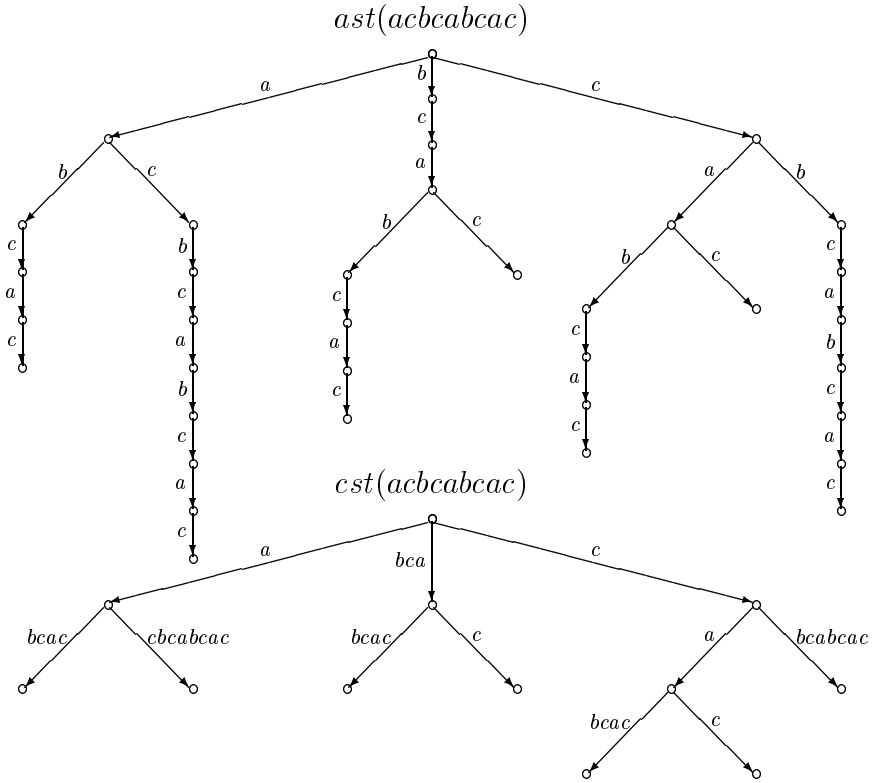
3.3 Suffix Trees

Suffix trees allow an efficient access to all subwords of a string. We define them as a special form of $\mathcal{A}^+$-trees:

Definition 3.3.1 A *suffix tree* for $x \in \mathcal{A}^*$ is an $\mathcal{A}^+$-tree T such that

$$words(T) = \{w \mid w \text{ is a subword of } x\}.$$

Figure 3.3: The Atomic and the Compact Suffix Tree for the String *acbcabcac*



The *atomic* suffix tree for x is denoted by $ast(x)$. The *compact* suffix tree for x is denoted by $cst(x)$. □

Example 3.3.2 Figure 3.1 shows the compact suffix tree for *ccabca*. Figure 3.3 shows the atomic and the compact suffix tree for the string *acbcabcac*. □

$ast(x)$ and $cst(x)$ are uniquely determined by the words occurring in them.

3.3.1 Applications

Suffix trees were first considered by Weiner [Wei73]. The original motivation for Weiner was to transmit or store a message with excerpts from a main string in minimum space or time. It became soon apparent that suffix trees are ideally suited for other, almost straightforward, applications.

- If T is a suffix tree for x, then w is a subword of x if and only if w occurs in T. The latter condition is checked in $\mathcal{O}(|\mathcal{A}| \cdot |w|)$ steps by traversing T down from the *root*, directed by the characters in w. This is useful for multiple exact searches of different strings in a fixed text.
- A leaf in a suffix tree for x corresponds to a suffix of x. Hence, with a suitable annotation of the suffix tree it is simple to find the first or the last end position of w

in x in $\mathcal{O}(|\mathcal{A}| \cdot |w|)$ steps. Similarly, one can compute *all* positions in x where w ends. In chapter 5 we describe this application in detail.

- The branching nodes in a suffix tree for x correspond to the right-branching subwords of x. This property is used in several algorithms for detecting structural patterns, like repeats [Apo85], squares [AP83, Apo85, Cro86, Kos94], and palindromes [Apo85].

- Several authors [RPE81, CWM84, Apo85, FG89] show that suffix trees are a suitable data structure for linear time sequential data compression techniques.

- Some special problems on strings can elegant and efficiently be solved with suffix trees. Among these are the computation of the longest common substring of two strings x and y in $\mathcal{O}(|x|+|y|)$ steps [Apo85], the test for unique decipherability of codes [Rod82], the statistics without overlap problem [AP85, Apo85] and the computation of the shortest unique subword of a string [Pow89].

- Chang and Lawler [CL90] have devised an exact string searching algorithm that traverses the suffix tree of the pattern. We consider this algorithm in detail in section 4.5.

- Recently, Bieganski et al. [BRCR94] have described a basic set of suffix tree operations to be used in a system for biological sequence analysis. In addition to the operations supported by our machinery, the authors outline a new suffix tree based method to compute alignments. Unfortunately, the presentation of this method is quite informal, and efficiency results remain unclear.

- In section 4.4.1 we give a new linear time algorithm which uses the suffix tree of the pattern to compute the functions used by the Boyer-Moore Algorithm [BM77].

- In [EH88] a new string distance measure is described which is computable in linear time using suffix trees. A similar efficiency result is achieved for the q-gram distance [Ukk92a]. For details see sections 3.9.2 and 3.9.3.

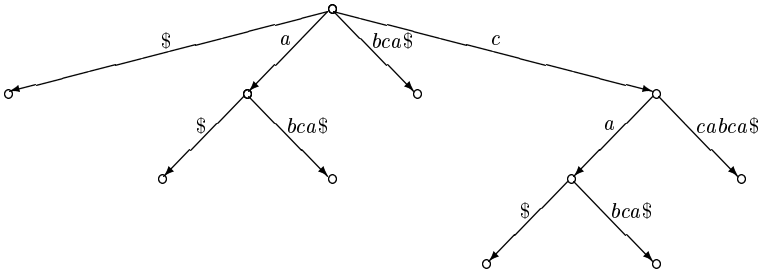
- The suffix tree of the pattern can be used to efficiently determine the longest prefix of a string which is also a subword of the pattern. With this property suffix trees have become a very important data structure in several approximate string searching algorithms [LV88, LV89, GG88, JTU91, Ukk92a, UW93, Mye94a, CL94]. We consider some of these algorithms in chapter 6.

3.3.2 Space Requirements

Note that for the efficiency of locating a subword of x it does not matter which kind of suffix tree for x we use. However, the space requirement for atomic and compact suffix trees differ in an order of magnitude:

- $ast(x)$ is the largest suffix tree for x. If $x \in \mathcal{A}^n$, then $ast(x)$ has $\mathcal{O}(n^2)$ nodes in the worst case (take $x = a^i b^i$, $i = n/2$). Isomorphic subtrees can be shared [CS85]. Sharing brings the space requirements down to $\mathcal{O}(n)$. However, subtree sharing may be impossible, when nodes are to be annotated with extra information.

Figure 3.4: The Compact Suffix Tree for *ccabca*\$



- $cst(x)$ is the smallest suffix tree for x. If $x \in \mathcal{A}^n$, then $cst(x)$ has $\mathcal{O}(n)$ nodes, as all inner nodes are branching, and there are at most n leaves. Since the edge labels are subwords of x, they can be represented in constant space by a pair of pointers into x. This is necessary to achieve a theoretical worst case bound of $\mathcal{O}(n)$. In practice, this is quite a delicate choice of representation in a virtual memory environment. Traversing the tree and reading the edge labels will create random-like accesses into x, and can lead to paging problems. This phenomenon is discussed in [GK95a] in more detail.

Besides atomic and compact suffix trees, some authors also discuss "position trees" [KBG87, MR80, GK94]. These are suffix trees of an intermediate level of compactness, that need $\mathcal{O}(n^2)$ space in the worst case and $\mathcal{O}(n \cdot \log n)$ space in the expected case [AS92]. We do not consider position trees in the thesis.

3.3.3 The Sentinel Character

If w is a nested suffix of x, then a suffix tree for x does not contain a leaf $\overline{w}$ (see Example 3.3.4 below). It is often convenient to add to x a *sentinel character*, say \$, that does not occur in x. Then $x\$$ has no nested suffix, except for the empty string. This implies the following lemma:

Lemma 3.3.3 Let w be a string and T be a suffix tree for $x\$$. Then w ends at position j in x if and only if $j = |x| - |u|$ for some string u such that $\overline{wu\$} \in leaves(T)$.

Proof Suppose w ends at position j in x. Then there is a string u such that $x \int wu$ and $j = |x| - |u|$. Hence, $x\$ \int wu\$$ which implies $\overline{wu\$} \in leaves(T)$. To show the opposite direction, suppose $\overline{wu\$} \in leaves(T)$ and let $j = |x| - |u|$. Then $x\$ \int wu\$$ and therefore $x \int wu$. Hence, w ends at position j in x. □

Example 3.3.4 The compact suffix tree for *ccabca*\$ is shown in Figure 3.4. The string c ends in *ccabca* at the positions $6-1=5$, $6-4=2$, and $6-5=1$. Notice that the correspondence stated in Lemma 3.3.3 does not hold without the sentinel character: Consider the compact suffix tree for *ccabca* as shown in Figure 3.1. The subtree $cst(ccabca)_{\overline{c}}$ has only two leaves, although c occurs three times in x. □

With respect to suffix links the sentinel character has the following effect:

Lemma 3.3.5 In $T = cst(x\$)$ all suffix links are atomic.

Proof We must show that $\overline{w} \in nodes(T)$ whenever $\overline{aw} \in nodes(T)$. $\overline{aw}$ is either a branching node, or a leaf in T. Hence, aw is right-branching in $x\$$, or a non-nested suffix of $x\$$. But then the same holds for w, and so $\overline{w} \in nodes(T)$. □

Note that in $cst(x)$ the inner suffix links are atomic. For a leaf $\overline{aw}$ in $cst(x)$, w may be nested and not right-branching in x. So there is no node $\overline{w}$ in $cst(x)$ and we have a non-atomic suffix link $\overline{aw} \longrightarrow \overline{v}$ for some proper suffix v of w. Now suppose that aw is the longest suffix of x such that aw is not nested, and w is nested. Then all suffixes of w are nested. Hence, aw is the only suffix of x with the above property. That is, there is at most one non-atomic suffix link in each compact suffix tree. This property was remarked by Robert Giegerich.

Example 3.3.6 Consider $cst(ccabca)$ as shown in Figure 3.1: The suffix link $\overline{bca} \longrightarrow root$ for the leaf $\overline{bca}$ is not atomic since there is no node $\overline{ca}$ in $cst(ccabca)$. □

It often simplifies proofs and constructions considerably to assume the presence of the sentinel character. However, in contexts where x may be expanded to the right (for instance, during online constructions), the requirement for a unique final character does not make sense.

Since the suffix link of a leaf $\overline{aw}$ in a suffix tree T may not be atomic, it cannot be used to access $loc_T(w)$ efficiently. As a consequence we have defined the location of $\overline{aw}$ in terms of the edge $\overline{u} \xrightarrow{v} \overline{aw}$ leading to $\overline{aw}$ (cf. Definition 3.2.6). This allows the efficient access to $loc_T(w)$, using the atomic suffix link of the inner node $\overline{u}$.

3.4 Implementation

In this section, we show how to implement $\mathcal{A}^+$-trees, suffix trees, and locations. As a prerequisite we discuss some important design decisions concerning the representation of strings, edge labels and edge sets.

3.4.1 Strings

In chapter 2, we have briefly discussed the general differences of the imperative and functional programming paradigm: The absence of details concerning sequencing and memory management in functional programs leads to implementations that often seem to be quite unrelated to the corresponding imperative implementations. In our particular application field there is another very important source for the dissimilarity: In imperative programs strings are usually represented by arrays, whereas in the functional world lists are preferred. So do we.

Definition 3.4.1 Suppose that the characters of the alphabet $\mathcal{A}$ are represented by some type α. Strings are represented by lists of type $[\alpha]$. This is expressed by the type synonym

```
string α == [α]
```

The empty list represents the empty string. The list $(a : w)$ represents the string aw. □

The type $(string\ \alpha)$ is polymorphic. It can be used for strings over arbitrary alphabets. In particular, it is not restricted to finite alphabets. However, we will assume that α contains only finitely many values and that the comparison of α-values takes constant time.

The decision to use lists was not made because Miranda does not offer arrays. We rather feel that lists are often a more natural representation of sequential information. As an example consider string searching algorithms which typically processes an input string from left to right such that the access to the input characters is limited to a small sliding window. Imperative implementations of string searching algorithms usually abstract from this fact and assume that the input string is stored in an array such that at any time each character can be accessed in a constant number of steps (see [Sed88, CLR90, Aho90, GBY91, BY92, Ste94]). However, in several applications the input string is given online, and some buffering mechanism is necessary to obtain a space efficient algorithm (cf. [BM77]). In our functional language we get such a buffering mechanism for free if we use lists: The laziness and the memory management of the Miranda system guarantee that at any time only the needed part of the list representing the input string is stored.

Note that the use of lists does not have a negative effect on the asymptotic running time. In almost all cases we can organize the computation such that a required value is held locally. This means that only constant time operations are applied to lists. In the few places where other operations are used (for example, indexing into a list or determining the length of a list) the overall running time is dominated by other computations.

3.4.2 Edge Labels

Consider the compact suffix tree for some string x. To achieve linear space, it is necessary to represent each edge label w of $cst(x)$ in constant space. As w is a subword of x, we can represent it by a pair (i, j) of integers such that $w = x_i \ldots x_j$. This is the usual way, as employed in [Ukk93b], for example. However, in our framework this solution has two disadvantages:

- It requires to store x as an array.
- It is inconvenient, as one always has to explicitly refer to x, when reading an edge label. This means that x becomes a part of the data structure suffix tree.

Of course, one can use extension 1 to get rid of the first disadvantage. However, we adopt the alternative representation of [GK95a] which avoids both disadvantages:

Definition 3.4.2 A subword w of x is represented by a pair $(s, |w|)$ of type

```
subword α == (string α,num)
```

where s is a suffix of x such that w is a prefix of s. □

This representation allows a convenient and efficient access to the length and the characters of the label w. Note that the representation is not unique. However, our implementations yield compact suffix trees which are labeled with subwords $(s, |w|)$ such that s is the *longest* suffix of x of which w is a prefix. This constraint proved to be helpful when testing the different suffix tree constructions.

The left component s of the *subword*-representation is a lazy list, that is, an unevaluated list expression. It should be thought of as a pointer to s, rather than a copy of s. Therefore, the representation of subwords needs constant space.

To *advance* i characters in a subword, the prefix of length i is dropped, and the right component is updated. This takes $\mathcal{O}(i)$ time altogether.

```
advance::num→(subword α)→(subword α)
advance i (s,j) = (drop i s,j-i)
```

3.4.3 Edge Sets

The efficiency of most algorithms involving $\mathcal{A}^+$-trees heavily depends on how expensive it is to access for each $a \in \mathcal{A}$ an a-edge outgoing from a node $\overline{s}$. Therefore, it is necessary to have a closer look on how the edges of an $\mathcal{A}^+$-tree can be stored. There are four basic techniques:

Arrays We can associate with each node an array of pointers, with one pointer for each $a \in \mathcal{A}$. This leads to a constant access time, but is prohibitive in size for large alphabets. It is only reasonably if almost every node has an a-edge for each $a \in \mathcal{A}$.

Ordered lists We can store the l outgoing edges for a node in an ordered list. Such a representation is simple and space efficient, but the costs to access a certain edge is proportional to l.

Binary search trees We can store the l outgoing edges of a node in a binary search tree [Knu73]. This is space efficient and the access time is proportional to $\log_2 l$. However, it leads to an overhead which only pays off if the average number of outgoing edges at each node is very large.

Hash tables We can use a hash table implementing a function $hash : nodes(T) \times \mathcal{A} \to nodes(T)$ such that $hash(\overline{s}, a) = \overline{saw}$ if and only if there is an edge $\overline{s} \xrightarrow{aw} \overline{saw}$ in T. McCreight [McC76] recommends this technique since he found it to be space efficient (in the average case) and reasonably speedy. However, the access time is not constant in general, due to the necessity of handling collisions.

Because every of the four technique has its disadvantages, sometimes a compromise is suitable. We can, for instance, store only the edges of the most frequently used nodes as an array. For the other nodes the outgoing edges can be encoded as an ordered list. This is the solution recommended in [AC75]. A comprehensive discussion of the different implementation techniques for suffix trees was recently given by Andersson and Nilsson [AN95]. For our functional implementation we use ordered lists, thereby following [GK95a].

Definition 3.4.3 The set of edges outgoing from a node of an $\mathcal{A}^+$-tree is represented by an ordered list. □

The main reason for this decision is that Miranda and other modern functional languages include polymorphic lists as a predefined data type with an efficient implementation and much notational convenience, like list comprehensions (see section 2.1.3). Using these polymorphic lists for representing the outgoing edges, leads to short, readable and efficient programs, that are hard to beat by an implementation based on one of the other techniques. For the efficiency of the implementations our decision leads to a factor l where l is the average number of edges outgoing from the visited nodes. Notice that in the worst case $l \in |\mathcal{A}|$.

3.4.4 Atomic $\mathcal{A}^+$-Trees

The design decisions discussed in the previous sections lead to the following representation of atomic $\mathcal{A}^+$-trees.

Definition 3.4.4 An atomic $\mathcal{A}^+$-tree T with suffix links and an S-annotation φ (cf. page 25) is implemented by the type $(tree\ \alpha\ \beta)$.

```
tree α β ::= N [edge α β] (tree α β) β | Undeftree
edge α β == (α,tree α β)
```

The elements of $\mathcal{A}$ and of S are represented by values of type α and β, respectively. Every node $\overline{s}$ in T is represented by an expression of the form $(N\ es\ link\ tag)$ such that the following holds.

1. es is a list containing for each edge $\overline{s} \xrightarrow{a} \overline{sa} \in edges(T)$ a pair $(a, node)$ such that $\overline{sa}$ is represented by the expression $node$. es is ordered according to the first component of the pairs.
2. If $\overline{s} \longrightarrow \overline{v} \in links(T)$, then $\overline{v}$ is represented by the expression $link$. If $\overline{s}$ is the $root$, then $link = Undeftree$. Since there is no other node with $link = Undeftree$, this convention allows us to identify the $root$ of an $\mathcal{A}^+$-tree.
3. tag is the representation of $\varphi(\overline{s})$. □

Note that the type $(tree\ \alpha\ \beta)$ is polymorphic. This enables us to use it as a unified representation of the various forms of $\mathcal{A}^+$-trees, which differ in the underlying alphabet, the degree of compression and the specific annotations for the nodes.

Example 3.4.5 The atomic $\mathcal{A}^+$-tree of Figure 3.1 is implemented by the following expression. Notice that a node $\overline{s} \neq root$ maps to the expression sN. The *tags* at the nodes are left undefined which is expressed by the predefined polymorphic value *undef*.

```
atree::tree char β
atree = root
        where root = N [('a',aN),('b',bN)] Undeftree undef
              aN = N [('b',abN)] root undef
              abN = N [('b',abbN)] bN undef
              abbN = N [] bN undef
              bN = N [] root undef
```

Note that some subexpressions in *atree* are shared so that it actually becomes a circular structure. For instance, bN occurs in three places: as a subexpression of *root*, of abN, and of $abbN$. In the latter two cases, bN is the expression the suffix link "points to". □

seledges and *seltag* select certain subexpressions of an expression $(N\ es\ link\ tag)$. *isleaf* tests if a node is a leaf. All three functions take constant time.

```
seledges::(tree α β)→[edge α β]
seledges (N es link tag) = es


seltag::(tree α β)→β
seltag (N es link tag) = tag


isleaf::(tree α β)→bool
isleaf (N es link tag) = (es = [])
```

3.4.5 Compact $\mathcal{A}^+$-Trees

Due to the polymorphism, we can reuse the type $(tree\ \alpha\ \beta)$. By instantiating the type variable α with $(subword\ \alpha)$ we obtain the type $(ctree\ \alpha\ \beta)$ for compact $\mathcal{A}^+$-trees. Similarly a type is introduced which represents edges that are labeled by subwords.

```
ctree α β == tree (subword α) β
cedge α β == edge (subword α) β
```

According to Definition 3.4.4, the nodes of an atomic $\mathcal{A}^+$-tree are represented by an expression of the form $(N\ es\ link\ tag)$. The like holds for the nodes of a compact $\mathcal{A}^+$-tree: The only difference is that the edge labels for the compact form are subwords.

Example 3.4.6 $cst(ccabca)$ (see Figure 3.1) is implemented by the following expression.

```
actree::ctree char β
actree = root
         where root = N [(("abca",4),abcaN),
                         (("bca",3),bcaN),
                         (("ccabca",1),cN)] Undeftree undef
               abcaN = N [] bcaN undef
               bcaN = N [] root undef
               cN = N [(("abca",4),cabcaN), (("cabca",5),ccabcaN)] root undef
               cabcaN = N [] abcaN undef
               ccabcaN = N [] cabcaN undef
```

□

3.4.6 Locations

The implementation of locations is straightforward: For node locations we introduce a constructor *LocN* and for edge locations a constructor *LocE*. Since the strings v and w in the edge location $(\overline{u}, v, w, \overline{uvw})$ are subwords of the same string, we represent them in the usual way by the type $(subword\ \alpha)$. This means that a location takes constant space.

```
location α β
  ::= LocN (ctree α β) | LocE (ctree α β) (subword α) (subword α) (ctree α β)
```

The following function returns the *ceiling* of a location.

```
ceiling::(location α β)→ctree α β
ceiling (LocN node) = node
ceiling (LocE node v w node') = node'
```

To determine if a location corresponds to the *root* and to a leaf, we use the functions *rootloc* and *leafloc*, respectively. *rootloc* exploits the fact that the *root* is the unique node with an undefined suffix link. *leafloc* makes use of the fact that a string corresponds to a leaf if and only if its location is of the form $(\overline{u}, v, \varepsilon, \overline{uv})$.

```
rootloc::(location α β)→bool
rootloc (LocN (N es Undeftree tag)) = True
rootloc loc = False

leafloc::(location α β)→bool
leafloc (LocE node v (w,0) node') = True
leafloc loc = False
```

The function *getloc* (see Definition 3.2.7) is implemented below. Note that the first argument of *getloc* is restricted to node locations. More precisely, if $\overline{s}$ is a node in T and w is a string such that $sw \in words(T)$ then $(getloc\ \overline{s}\ w)$ returns $loc_T(sw)$ in $\mathcal{O}(l \cdot |w|)$ time where l is the average number of edges outgoing from the nodes visited by *getloc*.

```
getloc::(ctree α β)→(subword α)→(location α β)
getloc node (w,0) = LocN node
getloc (N es link tag) (a:w,j)
  = LocE (N es link tag) (v,j) (advance j (v,i)) node, if isleaf node ∨ j < i
  = getloc node (advance i (a:w,j)),                  otherwise
    where ((v,i),node) = hd [((c:u,i),node) | ((c:u,i),node)←es; a = c]
```

Note that the definition of *getloc* becomes more efficient if we replace $(N\ es\ link\ tag)$ by an identifier, say $node'$, and then select es form $node'$. This is because reusing $node'$ saves the reconstruction of the expression $(N\ es\ link\ tag)$ against which $node'$ is matched. We have not done this optimization and will not do it in similar cases, for the following reasons. First, we think that an implementation using pattern matching is better to read. Second, the reconstruction of the matched expression does not have an influence on the asymptotic

efficiency. In Haskell and Clean, the function *getloc* would be implemented with *as-patterns*. These allow to give a name to a pattern, for use on the right hand side. In this way, readability and efficiency is achieved at the same time.

The function *scanprefix* (see Definition 3.2.9) is implemented as follows:

```
scanprefix::(location α β)→(string α)→(location α β,string α)
scanprefix loc w = (loc',w')
                   where (vnodes,loc',w') = scanprefix' loc w
```

scanprefix' is a generalization of *scanprefix*. It additionally computes the list *vnodes* of nodes visited during a traverse from *loc* to *loc'*. If *loc'* is a node location, then it is not included in *vnodes*. The laziness guarantees that *vnodes* is not evaluated if *scanprefix* is called. For implementing *scanprefix'*, we need a function *lcp* that returns the length of the longest common prefix of two strings u and v.

```
lcp::(string α)→(string α)→num
lcp u v = #takewhile (True=) [c=a | (c,a)←zip2 u v]

scanprefix'::(location α β)→(string α)→([ctree α β],location α β,string α)
scanprefix' loc [] = ([],loc,[])
scanprefix' (LocN node) (a:w)
  = down [((c:u,i),node') | ((c:u,i),node')←seledges node; a = c]
    where down [] = ([],LocN node,a:w)
          down (((c:u,i+1),node'):es)
            = ([node],loc',drop k w), if isleaf node' ∨ k < i
            = (node:vnodes,loc,w'),   otherwise
              where k = lcp u w,          if isleaf node'
                      = lcp (take i u) w, otherwise
                    loc' = LocE node (c:u,1+k) (advance k (u,i)) node'
                    (vnodes,loc,w') = scanprefix' (LocN node') (drop k w)
scanprefix' loc w
  = ([],loc',drop k w),                  if isleaf node' ∨ k < i
  = scanprefix' (LocN node') (drop k w), otherwise
    where LocE node (s,l) (u,i) node' = loc
          k = lcp u w,          if isleaf node'
            = lcp (take i u) w, otherwise
          loc' = LocE node (s,l+k) (advance k (u,i)) node'
```

The first equation for *scanprefix'* handles the case where the string to be scanned is empty. The third and fourth deal with edge locations. Node locations are handled in the second equation using a function *down* which is applied to the list of a-edges outgoing from *node*. The first equation for *down* covers the case where there is no such a-edge. The second equation corresponds to the case where the a-edge leads to a leaf or the edge label of the a-edge is not a prefix of aw. The third equation covers the other cases. Notice that $(scanprefix\ (loc_T(s))\ w)$ computes the result $(loc_T(su), v)$ in $\mathcal{O}(l \cdot |u|)$ steps, where l is the average number of edges outgoing from the visited nodes.

The function *scanprefix'* and even the special instance *scanprefix* are very general. Most programs involving compact $\mathcal{A}^+$-trees are based one of these functions.

The implementation of *linkloc* can almost literally be transfered from Definition 3.2.14.

```
linkloc::(location α β)→(location α β)
linkloc (LocN (N es link tag)) = LocN link
linkloc (LocE (N es link tag) av w node')
  = getloc (N es link tag) (advance 1 av), if link = Undeftree
  = getloc link av,                        otherwise
```

Note that *linkloc* exploits the fact that the *root* is the unique node for which the *link*-value is set to *Undeftree*.

linkloc is often applied iteratively. Therefore, we introduce a function *suffixlocs*. In particular, $(suffixlocs\ f\ (loc_T(s)))$ computes the list of locations of all suffixes of s and applies f to $loc_T(\varepsilon) = root$. If we instantiate f by the function $(:[\,])$, then we obtain a list which includes the *root*. If we instantiate f by the function $(const\ [\,])$, then we obtain a list without the *root*.

```
suffixlocs::(location α β→[location α β])→(location α β)→[location α β]
suffixlocs f loc = f loc,                          if rootloc loc
                 = loc:suffixlocs f (linkloc loc), otherwise
```

3.5 Suffix Tree Algorithms

There are two simple intuitive approaches to suffix tree construction. One is imperative by nature. It successively inserts the suffixes of x by updating an initially empty tree. This approach is used as a starting point for the derivation of Weiner's [Wei73], McCreight's [McC76] and also Ukkonen's [Ukk93b] method. Linear running time is achieved by using suffix links as "shortcuts" to certain locations in the tree.

An alternative approach centers on the data structure, that is, the suffix tree. It first determines the outgoing edges of the *root*, and then constructs the subtrees recursively in a top-down manner. No updates of the tree are necessary, and so this approach is declarative by nature. In particular, it is well-suited for a purely functional implementation. Unfortunately, the declarative approach leads to a quadratic worst case running time. However, the performance in practice is very good. Giegerich and Kurtz [GK94, GK95a] were the first to describe and analyze the declarative approach in detail. They called it lazy suffix tree algorithm. A short remark in [AN95] suggests that Andersson and Nilsson have independently discovered the virtues of this algorithm.

In this section we present three algorithms for constructing compact suffix trees. The lazy algorithm, Ukkonen's online algorithm and McCreight's algorithm. These are discussed in a similar way in [GK95a]. However, we use a slightly different notation and make some technical details explicit which are left open in [GK95a]. This leads to a more compact and concise declarative description. It provides the basis for proving the correctness of Ukkonen's and McCreight's algorithm. At least for the former we provide the first detailed correctness proof. We do not explain Weiner's algorithm because it is quite complicated and of no practical virtue, due to its space consumption. For a modern exposition of Weiner's algorithm we refer the reader to [LO94] or [GK95b]. The latter work especially clarifies the relation between the three linear time suffix tree algorithms.

3.5.1 The Lazy Suffix Tree Algorithm

Following [GK95a] we call a suffix tree algorithm (potentially) lazy when it constructs the suffix tree for x from the *root* towards the leaves. This has the advantage that the construction phase may be interleaved with tree traversal. Paths of the suffix tree need to be constructed only when being traversed for the first time. This kind of incrementality is achieved for free when implementing the lazy algorithm in a lazy language (see section 3.6.1). It can be simulated in an eager language by explicit synchronization between construction and (all) traversal routines (see [GKS95]).

As stated in [GK95a], the idea of the lazy algorithm is to group suffixes according to their first character and to take the longest common prefix of each group as the edge label. The following definition makes this description more precise.

Definition 3.5.1 Let $S \subseteq \mathcal{A}^*$. An element $v \in S$ is *superfluous* in S if $S \setminus \{v\} \neq \emptyset$ and v is a prefix of each element in $S \setminus \{v\}$. The functions $grouplcp : \mathcal{P}(\mathcal{A}^*) \to \mathcal{A}^* \times \mathcal{P}(\mathcal{A}^*)$ and $group : \mathcal{P}(\mathcal{A}^*) \to \mathcal{P}(\mathcal{A}^* \times \mathcal{P}(\mathcal{A}^*))$ are specified as follows:

- $grouplcp(S) = (w, S')$ if w is the longest string such that $\{ws \mid s \in S'\} = \{v \in S \mid v \text{ is not superfluous in } S\}$.
- For each $a \in \mathcal{A}$, $(aw, S'') \in group(S)$ if and only if $S' \neq \emptyset$ and $(w, S'') = grouplcp(S')$ where $S' = \{s \mid as \in S\}$. □

Note that superfluous strings result from nested suffixes.

Example 3.5.2 Let $x = acbcabcac$. Suppose

$S_1 = \{s \in \mathcal{A}^* \mid x \int cs\} = \{bcabcac, abcac, ac, \varepsilon\}$,
$S_2 = \{s \in \mathcal{A}^* \mid x \int acs\} = \{bcabcac, \varepsilon\}$,
$S_3 = \{s \in \mathcal{A}^* \mid x \int cbs\} = \{cabcac\}$,
$S_4 = \{s \in \mathcal{A}^* \mid x \int bs\} = \{cabcac, cac\}$.

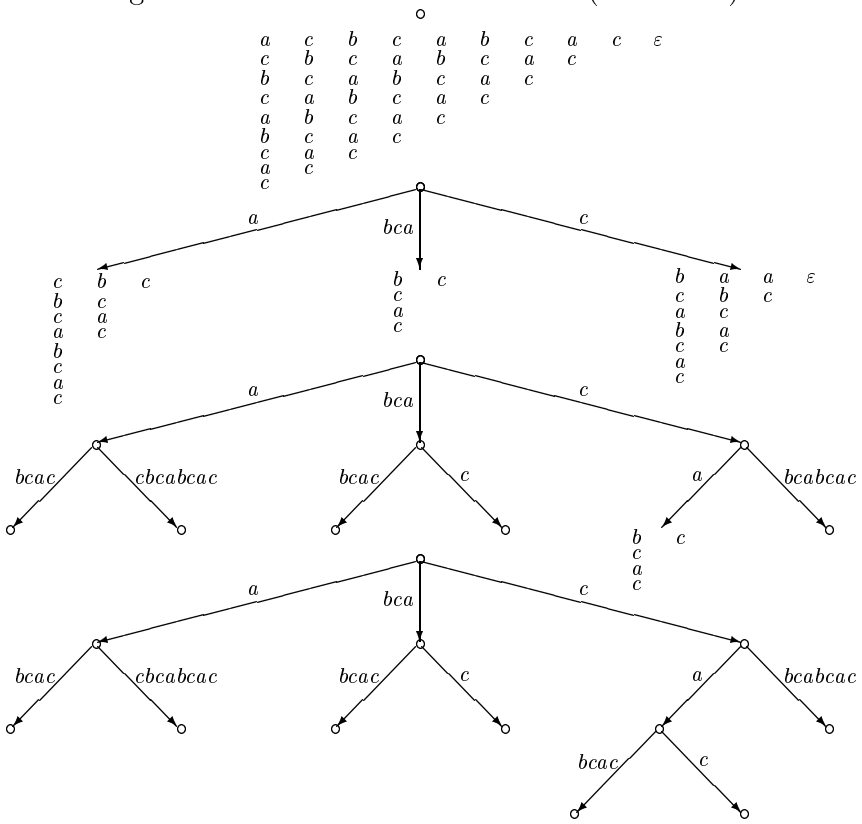
Note that ε is superfluous in S_1 and S_2. It results from the nested suffixes c and ac. We get the following:

$grouplcp(S_1) = (\varepsilon, \{bcabcac, abcac, ac\})$,
$grouplcp(S_2) = (bcabcac, \{\varepsilon\})$,
$grouplcp(S_3) = (cabcac, \{\varepsilon\})$,
$grouplcp(S_4) = (ca, \{bcac, c\})$. □

Let S be the set of suffixes of x. The lazy algorithm constructs $cst(x)$ directly from S. In particular, it creates for each $(aw, S'') \in group(S)$ an edge from the *root* labeled aw. Such an edge leads to a subtree recursively constructed from S''. The recursion terminates with a leaf edge when the set of suffixes becomes unitary. Figure 3.5 exemplifies how $cst(acbcabcac)$ is constructed.

Suffix trees imply a lexicographic ordering of all suffixes of a text. So it is easy to read the suffix array of [MM93a] from the suffix tree. In this sense the lazy algorithm constructs suffix arrays in a top-down (and left to right) fashion.

Figure 3.5: The Construction of $cst(acbcabcac)$



Let $l = |\mathcal{A}|$. According to [GK95a], the running time of the lazy algorithm is determined by the number of characters read from all suffixes, and the number of operations per character read. The sum of suffix lengths is $(n \cdot (n+1)/2)$. For $x = a^{n-1}\$$, all suffixes except for the longest are read to the last character. The grouping of suffixes can be implemented by counting sort [CLR90]. This leads to a space consumption of $\mathcal{O}(l+n)$, but avoids iteration over the characters in $\mathcal{A}$. The worst case is therefore $\mathcal{O}(n^2)$ for $x = a^{n-1}\$$. The expected length of the longest repeated subword is $\mathcal{O}(\log_l n)$ (see [AS92]). Since no suffix is read beyond the point where it becomes unique, the average case efficiency is $\mathcal{O}(n \cdot \log_l n)$. l does not occur as a factor in the $\mathcal{O}$-terms above. This is a property that the lazy algorithm shares with the suffix array algorithm of [MM93a]. All other known suffix tree algorithms must use more complicated data structures to reduce the alphabet factor (see section 3.4.3).

The measurements in [GK95a] show that the lazy algorithm is practically linear and becomes faster for larger alphabets. This is due to the locality behavior which determines the performance of the memory subsystem and indirectly affects the running time. The lazy algorithm has optimal locality on the tree data structure. Once a subtree is completed, it is not accessed again. In principle, more than a "current path" in the tree need not be in memory. With respect to text access, the lazy algorithm also behaves very well. For each subtree, only the corresponding suffix-rests are accessed. At a certain tree level, the number of suffixes considered will be smaller than the number of available cache entries. As these suffixes are read sequentially, practically no further cache misses will occur. This point is reached earlier when the branching degree of the tree nodes is higher, since the suffixes split up more quickly. Hence, the locality of the lazy algorithm improves for larger values of l.

3.5.2 Ukkonen's Online Suffix Tree Algorithm

In this section, we review Ukkonen's online suffix tree algorithm [Ukk93b]. While Ukkonen derives his algorithm in an operational style using the atomic suffix tree as an intermediate step, we follow [GK95a] and give a declarative presentation based on properties of suffixes. However, our presentation improves on [GK95a] in two aspects: On the one hand, we use locations which simplifies the description. On the other hand, we treat the delayed setting of the suffix links explicitly. This is quite intricate, but necessary for deriving the implementation in section 3.6.5.

The online algorithm generates a series of compact suffix trees for longer and longer prefixes of a string. The initial step of the algorithm is trivial: $cst(\varepsilon)$ is just the *root* with no edges. Suppose that xay is an arbitrary string for which we want to construct the compact suffix tree. Assume that we have read the prefix xa and that $cst(x)$ is already constructed. At this point of the algorithm we have not "seen" anything of y. Therefore, let us call xa the *visible part*, and y the *hidden part* of xay. In terms of functional programming, y can be thought of as a lazy list, that is, an unevaluated list expression. As the online algorithm proceeds, more and more characters of y become visible, that is, more and more of y is evaluated.

The crucial point of the online algorithm is the step from $cst(x)$ to $cst(xa)$. Since $cst(xa)$ must represent all subwords of xa, we consider all new subwords of xa, that is, all those not occurring in $cst(x)$. Every new subword of xa is obviously a non-empty suffix of xa, that is, it is of the form sa where s is a suffix of x. Note that $\overline{sa}$ has to be a leaf in $cst(xa)$, since otherwise sa would be a subword of x and hence occur in $cst(x)$.

If $\overline{s}$ is a leaf in $cst(x)$, then the leaf edge $\overline{u} \xrightarrow{w} \overline{s}$ of $cst(x)$ gives rise to the leaf edge $\overline{u} \xrightarrow{wa} \overline{sa}$ in $cst(xa)$. Therefore, the labels of all leaf edges in $cst(x)$ have to grow by the character a. To accomplish this growth all at once in constant time, Ukkonen [Ukk93b] treats the leaf edges as open edges. The idea is that a label of a leaf edge in $cst(xa)$ is of the form way for some suffix w of x. Like xay we can also divide way into a visible and a hidden part.

Definition 3.5.3 An *open edge* in $cst(xa)$ is a leaf edge with an edge label way, that represents the suffix wa of xa. wa is the *visible part* of way, and y the *hidden part*. For Ukkonen's online algorithm all leaf edges are open edges. □

Our definition of open edges is tailored for the implementation of edge labels by the type (*subword* α) (see Definition 3.4.2). If the edge labels are implemented by a pair of integers instead, then the solution introduced in [Ukk93b] is recommended: An open edge is labeled by the pair (i, ∞), where the symbol ∞ means that the edge is open to grow. This solution is equivalent to ours.

Due to open edges, the labels of leaf edges grow while more and more characters become visible. Thus, to insert a new suffix sa into $cst(x)$, nothing must be done when $\overline{s}$ is a leaf. Hence, we only consider the complementary case, when $\overline{s}$ is not a leaf in $cst(x)$, or equivalently s is a nested suffix of x.

Definition 3.5.4 [GK95a] A suffix sa of xa is *relevant* if s is a nested suffix of x and sa is not a subword of x. □

The step from $cst(x)$ to $cst(xa)$ now means the following: Insert all relevant suffixes sa of xa into $cst(x)$. To make this description more precise, we study some properties of relevant suffixes. In particular, we show that the relevant suffixes of xa form a contiguous segment of the list of all suffixes of xa, whose bounds are marked by "active suffixes":

Definition 3.5.5 [GK95a] The *active suffix* of x, denoted by $\alpha(x)$, is the longest nested suffix of x. □

The notion active suffix corresponds to the notion active point introduced in [Ukk93b].

Example 3.5.6 [GK95a] Consider the string *adcdacdad* and a list of columns, where each column contains the list of all suffixes of a prefix of this string. The relevant suffixes in each column are marked by the symbol ↓ and the active suffix is printed in bold face.

$\boldsymbol{\varepsilon}$	$\downarrow a$	ad	adc	$adcd$	$adcda$	$adcdac$	$adcdacd$	$adcdacda$	$adcdacdad$
	$\boldsymbol{\varepsilon}$	$\downarrow d$	dc	dcd	$dcda$	$dcdac$	$dcdacd$	$dcdacda$	$dcdacdad$
		$\boldsymbol{\varepsilon}$	$\downarrow c$	cd	cda	$cdac$	$cdacd$	$cdacda$	$cdacdad$
			$\boldsymbol{\varepsilon}$	$\boldsymbol{d}$	$\downarrow da$	dac	$dacd$	$dacda$	$dacdad$
				ε	$\boldsymbol{a}$	$\downarrow ac$	acd	$acda$	$acdad$
					ε	$\boldsymbol{c}$	$\boldsymbol{cd}$	$\boldsymbol{cda}$	$\downarrow cdad$
						ε	d	da	$\downarrow dad$
							ε	a	$\boldsymbol{ad}$
								ε	d
									ε

Lemma 3.5.7

1. For all suffixes s of x: s is nested $\iff |\alpha(x)| \geq |s|$.
2. For all suffixes s of x: sa is a relevant suffix of $xa \iff |\alpha(x)a| \geq |sa| > |\alpha(xa)|$.
3. $\alpha(xa)$ is a suffix of $\alpha(x)a$.
4. If $sa = \alpha(xa)$ and $\alpha(x)a \neq sa$, then s is a right-branching subword of x.

Proof

1. Routine.
2. sa is a relevant suffix of $xa \iff s$ is a nested suffix of x and sa is not a subword of $x \iff |\alpha(x)| \geq |s|$ and sa is not a nested suffix of $xa \iff |\alpha(x)a| \geq |sa|$ and $|sa| > |\alpha(xa)| \iff |\alpha(x)a| \geq |sa| > |\alpha(xa)|$.
3. Since both $\alpha(xa)$ and $\alpha(x)a$ are suffixes of xa, it suffices to show $|\alpha(x)a| \geq |\alpha(xa)|$. If $\alpha(xa) = \varepsilon$, then this is obviously true. Let $\alpha(xa) = wa$. Since wa is a nested suffix of xa, we have $uwav = x$ for some strings u and v. Hence, w is a nested suffix of x. Since $\alpha(x)$ is the longest nested suffix of x, we have $|\alpha(x)| \geq |w|$ and hence $|\alpha(x)a| \geq |wa| = |\alpha(xa)|$.

4. Suppose $sa = \alpha(xa)$ and $\alpha(x)a \neq sa$. Then there is a suffix csa of xa such that $|\alpha(x)a| \geq |csa| > |\alpha(xa)|$. From Statement 2 we know that csa is a relevant suffix of xa. That is, cs is a nested suffix of x, and csa is not a subword of x. Hence, there is a character $b \neq a$ such that csb is a subword of x. Since sa is a subword of x, too, s is a right-branching subword of x. □

Except for Statement 4, these properties were already proved in [GK95a]. By Statement 2 the relevant suffixes of xa are "between" $\alpha(x)a$ and $\alpha(xa)$. Hence, by Statement 3, $\alpha(xa)$ is the longest suffix of $\alpha(x)a$ that is a subword of x. Based on this fact, the step from $cst(x)$ to $cst(xa)$ can be described as follows:

> Take the suffixes of $\alpha(x)a$ one after the other by decreasing length and insert them into $cst(x)$, until a suffix is found which occurs in the tree and therefore equals $\alpha(xa)$.

This can formally be described by a function *naiveukkstep* which has three arguments: An arbitrary string ay, a compact $\mathcal{A}^+$-tree T and a string s such that $\alpha(x)a \sqsupset sa \sqsupset \alpha(xa)$ and $words(T) = words(cst(x)) \cup \{va \mid \alpha(x)a \sqsupset va \sqsupset sa, v \neq s\}$. In other words, the relevant suffixes va of xa that are longer than sa already occur in T.

Definition 3.5.8 The function *naiveukkstep* is defined as follows:

$$naiveukkstep(T, s, ay) = \begin{cases} (T, sa), & \text{if } occurs(loc, a) \\ (T[loc \leftarrow ay], s), & \text{else if } loc = root \\ naiveukkstep(T[loc \leftarrow ay], u, ay), & \text{otherwise} \end{cases}$$

where $loc = loc_T(s)$

$bu = s$ for some $b \in \mathcal{A}$ and some $u \in \mathcal{A}^*$

Lemma 3.5.9 $naiveukkstep(cst(x), \alpha(x), ay) = (cst(xa), \alpha(xa))$.

Proof By induction on $|sa| - |\alpha(xa)|$ one can show that $naiveukkstep(T, s, ay)$ returns $(cst(xa), \alpha(xa))$ (cf. Lemma 3.5.12). With $T = cst(x)$ and $s = \alpha(x)$ the claim follows. □

The crucial point in the function *naiveukkstep* is to determine the location of s in T. The easiest way to accomplish this is to follow the path for s down from the *root*, anew for each suffix s. This leads to a naive version of Ukkonen's algorithm, which is called *naiveOnline* in [GK95a]. The efficiency of *naiveOnline* is as follows:

Theorem 3.5.10 [GK95a] Let $x \in \mathcal{A}^n$ and $l = |\mathcal{A}|$. *naiveOnline* computes $cst(x)$ in $\mathcal{O}(l \cdot n^2)$ time in the worst case and in $\mathcal{O}(l \cdot n \cdot \log n)$ time in the expected case.

Proof There are $\mathcal{O}(n)$ nodes created. The path length to access each node is $\mathcal{O}(n)$ in the worst and $\mathcal{O}(\log n)$ in the expected case [AS92]. Selecting the suitable branch at each node introduces a factor l. □

An optimal functional program for *naiveOnline* is given in [GK95a]. Along the path from the *root* to $loc_T(s)$ the program de- and reconstructs the tree in $\mathcal{O}(l)$ steps for each node visited. Thus, it turns a local update into a global one with no effect on asymptotic efficiency.

For achieving linear running time, the suffix s of $\alpha(x)$ is represented in constant space by its location in T. A direct access to the location of the next suffix is provided by the function *linkloc*. This leads to a function *ukkstep*.

Definition 3.5.11 The function *ukkstep* is defined as follows:

$$ukkstep(T, L, ay, \overline{z}, loc) = \begin{cases} (T, L', getloc(loc, a)), & \text{if } occurs(loc, a) \\ (T', L', loc), & \text{else if } loc = root \\ ukkstep(T', L', ay, \overline{r}, linkloc(loc)), & \text{otherwise} \end{cases}$$

where $(T', \overline{r}) = T[loc \leftarrow ay]$

$$L' = \begin{cases} L, & \text{if } \overline{z} = \perp \\ L \cup \{\overline{z} \longrightarrow loc\}, & \text{else if } occurs(loc,a) \text{ or } \overline{r} = \perp \\ L \cup \{\overline{z} \longrightarrow \overline{r}\}, & \text{otherwise} \end{cases}$$

Note that for a new inner node $\overline{z}$ the suffix link $\overline{z} \longrightarrow \overline{r}$ cannot be set instantly, since $\overline{r}$ may not exist yet. However, $\overline{r}$ will be created in the next call to *ukkstep*. Therefore, $\overline{z}$ is taken as an argument of *ukkstep* and the setting of the suffix link is delayed until $\overline{r}$ is constructed. The following lemma proves the correctness of *ukkstep*.

Lemma 3.5.12 Suppose that for T, L, $\overline{z}$, and loc the following properties hold:

1. loc is the location of s in T for some s such that $\alpha(x)a \sqsupset sa \sqsupset \alpha(xa)$.

2. T is the compact $\mathcal{A}^+$-tree such that
$words(T) = words(cst(x)) \cup \{va \mid \alpha(x)a \sqsupset va \sqsupset sa, v \neq s\}$.

3. If $s = \alpha(x)$, then $L = innerlinks(T)$ and $\overline{z} = \perp$. Suppose $s \neq \alpha(x)$. Then there is a compact $\mathcal{A}^+$-tree T'' and a location loc'' in T'' such that $(T, \overline{z}) = T''[loc'' \leftarrow ay]$. If $\overline{z} = \perp$, then $L = innerlinks(T)$. Otherwise, $L = innerlinks(T'')$. This means that $\overline{z}$ is either set to $\perp$ or the suffix link for $\overline{z}$ is not already set.

Then $ukkstep(T, L, ay, \overline{z}, loc) = (cst(xa), innerlinks(cst(xa)), loc_{cst(xa)}(\alpha(xa)))$.

Proof By induction on $|sa| - |\alpha(xa)|$.

- Suppose $|sa| - |\alpha(xa)| = 0$. Then $sa = \alpha(xa)$, that is, $occurs(loc, a)$ is true. Hence, $ukkstep(T, L, ay, \overline{z}, loc) = (T, L', getloc(loc, a))$. Obviously, $T = cst(xa)$. Moreover, $getloc(loc, a) = getloc(loc_T(s), a) = loc_T(sa) = loc_T(\alpha(xa))$. If $\overline{z} = \perp$, then we have $L' = L = innerlinks(T)$. Otherwise, by Lemma 3.5.7, Statement 4, s is a right-branching subword of x. Thus, $loc = loc_T(s)$ is a branching node in T which means that $L' = L \cup \{\overline{z} \longrightarrow loc\} = innerlinks(T'') \cup \{\overline{z} \longrightarrow loc\} = innerlinks(T)$.

- Suppose $|sa| - |\alpha(xa)| > 0$ and $s = \varepsilon$. Then $\alpha(xa) = \varepsilon$ and therefore $loc = root$ and $occurs(loc, a)$ is false. Hence, $ukkstep(T, L, ay, \overline{z}, loc) = (T', L', loc)$, where $(T', \overline{r}) = T[loc \leftarrow ay]$. Obviously, $\overline{r} = \perp$, $T' = cst(xa)$, and $loc = root = loc_{T'}(\alpha(xa))$. If $\overline{z} = \perp$, then $L' = L = innerlinks(T) = innerlinks(T')$. Otherwise, $L' = L \cup \{\overline{z} \longrightarrow loc\} = innerlinks(T'') \cup \{\overline{z} \longrightarrow loc\} = innerlinks(T) = innerlinks(T')$.

- Suppose $|sa| - |\alpha(xa)| > 0$ and $s \neq \varepsilon$. Then $occurs(loc, a)$ is false and $loc \neq root$. Hence,

$$ukkstep(T, L, ay, \overline{z}, loc) = ukkstep(T', L', ay, \overline{r}, linkloc(loc)) \tag{3.1}$$

  where $(T', \overline{r}) = T[loc \leftarrow ay]$. It is easy to verify that the properties 1-3 hold for T', L', $\overline{r}$, and $linkloc(loc)$ considerably. Hence, we can apply the induction hypothesis and the right hand side of (3.1) reduces to $(cst(xa), innerlinks(cst(xa)), loc_{cst(xa)}(\alpha(xa)))$. This completes the proof. □

Ukkonen's online algorithm is specified by the function ukk:

$$\begin{aligned} ukk(T, L, \varepsilon, loc) &= (T, L) \\ ukk(T, L, ay, loc) &= ukk(T', L', y, loc') \\ &\quad \text{where } (T', L', loc') = ukkstep(T, L, ay, \bot, loc) \end{aligned}$$

Theorem 3.5.13 Let $x \in \mathcal{A}^n$, $l = |\mathcal{A}|$ and $T = cst(x)$. Then $ukk(cst(\varepsilon), \emptyset, x, root)$ returns $(T, innerlinks(T))$ in $\mathcal{O}(l \cdot n)$ time and $\mathcal{O}(n)$ space.

Proof The correctness of ukk follows from Lemma 3.5.12. The complexity proof carries over from [Ukk92b]. □

In section 3.6.5, we give a monadic implementation of Ukkonen's online algorithm which closely resembles the structure of the functions $ukkstep$ and ukk.

3.5.3 McCreight's Suffix Tree Algorithm

The suffix tree algorithm of McCreight is widely known. McCreight's presentation is semi-formal (see [McC76, page 266]) and commonly considered as rather complicated and difficult to grasp (see [Ukk92b, page 484]). Other authors describing McCreight's algorithm either closely follow the presentation in [McC76] (see [Ste94]) or they only briefly outline the main ideas (see [RPE81, FG89, CL94]). Recently, Giegerich and Kurtz [GK95a] have given a formal treatment of McCreight's algorithm. So do we. However, we use a different notation. This leads to a more compact description which maps to the implementation in a transparent and relatively straightforward way.

For this section we suppose that x is a string of length $n \geq 2$ whose last character is the sentinel character \$. We start the description by introducing some terminology.

Definition 3.5.14 Let s be a suffix of x.

- $T(s)$ is the compact $\mathcal{A}^+$-tree such that $words(T(s)) = \{u \mid x \sqsupset w \sqsupset s, u \sqsubset w\}$.
- A prefix u of s is *left-occurring* if $u \sqsubset w$ for some string $w \neq s$ and $x \sqsupset w \sqsupset s$.
- If $s \neq x$, then $head(s)$ is the longest left-occurring prefix of s. If $s = x$, then $head(s)$ is the empty string.
- $tail(s) = v$ if $s = head(s)v$. □

A left-occurring prefix is a nested prefix in the terminology of [GK95a]. We did not adopt the term nested prefix since it suggests symmetry to the notion of nested suffixes, which is not the case. The following lemma clearly states the relation between left-occurring prefixes and nested suffixes. The notions *head* and *tail* are adopted from [McC76]. The compact $\mathcal{A}^+$-tree $T(x_i \dots x_n)$ is denoted by T_i in [McC76].

Lemma 3.5.15 Let s be a non-empty suffix of x. Then the following holds.

- $tail(s) \neq \varepsilon$.
- If $x = ws$ for some w, then the string u is a left-occurring prefix of s if and only if u is a nested suffix of wu.
- If as is a suffix of x, then $head(s)$ is the longest prefix of s that occurs in $T(as)$.

Proof Obvious. □

The general structure of McCreight's algorithm is to construct $cst(x)$ by successively inserting the suffixes of x into an initially empty tree, from longest to shortest. More precisely, the algorithm constructs the sequence

$$cst(\varepsilon), T(x_1 \dots x_n), T(x_2 \dots x_n), \ \cdots \ , T(x_{n-1}x_n), T(x_n) = cst(x) \tag{3.2}$$

of compact $\mathcal{A}^+$-trees, of which only the first and the last one is a suffix tree.

The initial step of the algorithm is trivial: $T(x) = T(x_1 \dots x_n)$ is obtained from $cst(\varepsilon)$ by inserting the longest suffix x. Thus, $T(x)$ is the compact $\mathcal{A}^+$-tree with only one edge $root \xrightarrow{x} \overline{x}$. Let as be a suffix of x. For the step from $T(as)$ to $T(s)$ it is most important to compute $loc_{T(as)}(head(s))$ and $tail(s)$. Once we have done this we can simply construct $T(s)$ from $T(as)$ by an insert-operation.

Lemma 3.5.16 $T(s) = T(as)[loc_{T(as)}(head(s)) \leftarrow tail(s)]$ for all suffixes as of x.

Proof We have

$$\begin{aligned} words(T(s)) &= \{u \mid x \sqsupset w \sqsupset s, u \sqsubset w\} \\ &= \{u \mid x \sqsupset w \sqsupset as, u \sqsubset w\} \cup \{u \mid u \sqsubset s\} \\ &= words(T(as)) \cup \{u \mid head(s) \sqsubset u \sqsubset s, u \neq head(s)\} \\ &= words(T(as)) \cup \{head(s)w \mid w \sqsubset tail(s), w \neq \varepsilon\} \\ &= words(T(as)[loc_{T(as)}(head(s)) \leftarrow tail(s)]). \end{aligned}$$

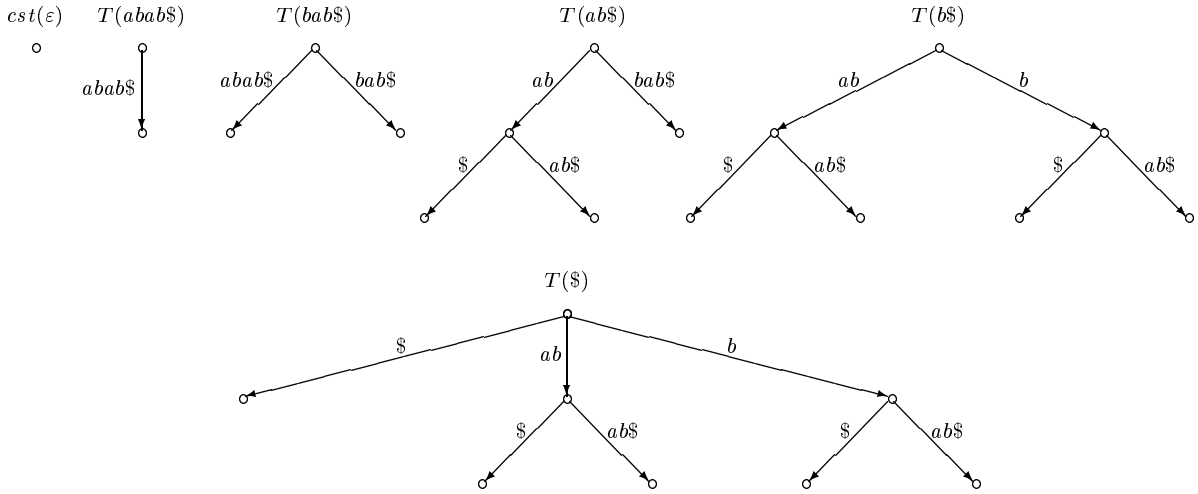
Note that the last equality follows from Lemma 3.2.12. □

Example 3.5.17 Let $x = abab\$$. Figure 3.6 shows the sequence of compact $\mathcal{A}^+$-trees constructed to obtain $cst(x)$. Moreover, we have

$$\begin{array}{ll} head(abab\$) = \varepsilon & tail(abab\$) = abab\$ \\ head(bab\$) = \varepsilon & tail(bab\$) = bab\$ \\ head(ab\$) = ab & tail(ab\$) = \$ \\ head(b\$) = b & tail(b\$) = \$ \\ head(\$) = \varepsilon & tail(\$) = \$ \end{array}$$

□

Figure 3.6: The Compact $\mathcal{A}^+$-Trees constructed to obtain $T(\$) = cst(abab\$)$



The easiest way to determine $loc_{T(as)}(head(s))$ and $tail(s)$ is to follow the path for s in $T(as)$ down from the *root*, until one "falls out of the tree" (as guaranteed by the sentinel character in x). In other words, we have $scanprefix(root, s) = (loc_{T(as)}(head(s)), tail(s))$. Using *scanprefix* to compute $loc_{T(as)}(head(s))$ and $tail(s)$, leads to a naive version of McCreight's algorithm. Giegerich and Kurtz [GK95a] call this algorithm *naiveInsertion* and show an efficiency of $\mathcal{O}(l \cdot n^2)$ in the worst case and of $\mathcal{O}(l \cdot n \cdot \log n)$ in the average case (l is the size of the alphabet). An optimal functional implementation of *naiveInsertion* is given in [GK95a]. Like *naiveOnline* it de- and reconstructs the tree during the scan from the *root*, turning the local updates into global ones without extra overhead.

To obtain a linear-time algorithm, $loc_{T(as)}(head(s))$ and $tail(s)$ must be computed in constant time (averaged over all steps). This is accomplished by exploiting the following relationships:

Lemma 3.5.18 Let as be a suffix of x. Suppose $head(as) = aw$ for some string w. Then the following holds:

1. There is a branching node $\overline{aw}$ in $T(as)$.
2. w is a prefix of $head(s)$.
3. $w = head(s)$ whenever there is no branching node $\overline{w}$ in $T(as)$.

Proof Since aw is a left-occurring prefix of as, we have $x \int awcv \int as = awdu$ for some characters c and d and some strings u and v such that $awcv \neq as$. Obviously, $d \neq c$, since otherwise awd would be a left-occurring prefix of as. As both awd and awc occur in $T(as)$, $\overline{aw}$ is a branching node in $T(as)$, that is, Statement 1 is proved. Moreover, $x \int wcv \int s = wdu$ which means that w is a left-occurring prefix of s. As $head(s)$ is the longest left-occurring prefix of s, w is a prefix of $head(s)$, which proves Statement 2. Since $wcv \neq s$, we can conclude $x \int wcv \int as$. Hence, wcv occurs in $T(as)$. Now suppose that there is no branching node $\overline{w}$ in $T(as)$. Then wd does not occur in $T(as)$. Hence, w is the longest prefix of $wdu = s$, which occurs in $T(as)$. Therefore, $w = head(s)$ which proves Statement 3. □

Note that McCreight explicitly mentions only the second property of Lemma 3.5.18 (see [McC76, Lemma 1]). Apostolico [Apo85, page 86] claims that all clever variations of suffix trees are built in linear time by resorting to similar properties. In fact, for Ukkonen's online algorithm (which was described several years after Apostolico has made his claim) we have such a property, too. It is stated in Lemma 3.5.7, Statement 2: The relevant suffixes of xa form a contiguous segment of the list of all suffixes of xa, whose bounds are marked by active suffixes.

The following function specifies a construction step in McCreight's algorithm.

Definition 3.5.19 The function *mccstep* is defined as follows:

$$
\begin{aligned}
&mccstep(T, L, loc, cy, \overline{z})\\
&\quad = (T', L', loc'', y', \overline{r})\\
&\qquad \text{where } loc' = linkloc(loc)\\
&\qquad\qquad (T', \overline{r}) = T[loc'' \leftarrow y']\\
&\qquad\qquad ((loc'', y'), L') = \begin{cases}
(scanprefix(loc, y), L), & \text{if } loc = root\\
(scanprefix(loc', cy), L), & \text{else if } loc \text{ is a node}\\
(scanprefix(loc', cy), L \cup \{\overline{z} \longrightarrow loc'\}), & \text{else if } loc' \text{ is a node}\\
((loc', cy), L \cup \{\overline{z} \longrightarrow \overline{r}\}), & \text{otherwise}
\end{cases}
\end{aligned}
$$

Lemma 3.5.20 Let as be a suffix of x and T'' be the compact $\mathcal{A}^+$-tree preceding $T(as)$ in the sequence (3.2), i.e., $(T(as), \overline{z}) = T''[loc_{T''}(head(as)) \leftarrow tail(as)]$ (see Lemma 3.5.16). Then

$$mccstep(T(as), innerlinks(T''), loc_{T''}(head(as)), tail(as), \overline{z})$$

returns

$$(T(s), innerlinks(T(as)), loc_{T(as)}(head(s)), tail(s), \overline{r}),$$

where $(T(s), \overline{r}) = T(as)[loc_{T(as)}(head(s)) \leftarrow tail(s)]$.

Proof

1. If $loc = root$, then $head(as) = \varepsilon$. Therefore, $as = tail(as) = cy$ which implies $y = s$. Hence, $(loc'', y') = scanprefix(loc, y) = (loc_{T(as)}(head(s)), tail(s))$. Moreover, since loc is a node location, $\overline{z} \in inner(T'')$. That is, there is no suffix link to be set. Thus, $L' = L = innerlinks(T'') = innerlinks(T(as))$.

2. If $loc \neq root$, then $head(as) = aw$ for some string w. By Lemma 3.5.18, Statement 2, w is a prefix of $head(s)$. By definition $loc_{T''}(aw) = loc$. Let $loc' = linkloc(loc)$. loc' is defined since only the inner suffix links of T'' are used to compute it. Hence, $loc' = linkloc(loc) = loc_{T''}(w) = loc_{T(as)}(w)$ by Lemma 3.2.15.

   (a) If loc is a node, then so is loc' and we obtain $(loc'', y') = scanprefix(loc', cy) = (loc_{T(as)}(head(s)), tail(s))$. Moreover, $\overline{z} \in inner(T'')$. Thus, there is no suffix link to be set and we obtain $L' = L = innerlinks(T'') = innerlinks(T(as))$.

   (b) If loc is an edge location, then $\overline{z} \in inner(T(as)) \setminus inner(T'')$. That is, the suffix link for $\overline{z}$ has not been set.

   - If loc' is a node, then we have $loc' = \overline{w}$ and $L' = L \cup \{\overline{z} \longrightarrow loc'\} = innerlinks(T'') \cup \{\overline{z} \longrightarrow loc'\} = innerlinks(T(as))$. Moreover, $(loc'', y') = scanprefix(loc', cy) = (loc_{T(as)}(head(s)), tail(s))$.

- If loc' is an edge location, then by Lemma 3.5.18, Statement 3, we obtain $w = head(s)$ and hence $(loc'', y') = (loc', cy) = (loc_{T(as)}(head(s)), tail(s))$. Moreover, we conclude $L' = L \cup \{\overline{z} \longrightarrow \overline{r}\} = innerlinks(T'') \cup \{\overline{z} \longrightarrow \overline{r}\} = innerlinks(T(as))$, where the node $\overline{r} = \overline{w}$ will be created with $T(s)$. □

McCreight's algorithm is specified by the function mcc.

$$mcc(T, L, loc, cy, \overline{z}) = \begin{cases} (T, L), & \text{if } y = \varepsilon \text{ and } loc = root \\ mcc(mccstep(T, L, loc, cy, \overline{z})), & \text{otherwise} \end{cases}$$

Theorem 3.5.21 Let $x \in \mathcal{A}^n$, $l = |\mathcal{A}|$ and $T = cst(x)$. Then $mcc(T(x), \emptyset, root, x, \bot)$ returns $(T, innerlinks(T))$ in $\mathcal{O}(l \cdot n)$ time and $\mathcal{O}(n)$ space.

Proof The correctness follows from Lemma 3.5.20. The complexity arguments carry over from [McC76]. □

Before we consider implementation issues, we should remark that Ukkonen's and McCreight's suffix tree algorithms are very closely related, although they are based on rather different intuitive ideas. Based on a declarative description, Giegerich and Kurtz [GK95b] have shown the following:

- Ukkonen's algorithm can be transformed into McCreight's algorithm by a modification of its control structure, leaving the sequence tree insertion operations invariant.
- The modification is a slight optimization. Under a fair implementation, of the related data structures, it will give McCreight's algorithm a minor efficiency advantage over Ukkonen's algorithm, on every possible input. This is confirmed by the measurements in [GK95a].
- The modification sacrifices the online property. McCreight's algorithm will always read ahead of Ukkonen's algorithm in x.

The result of Giegerich and Kurtz is one more example which shows that a declarative view often leads to new interesting insights about algorithms.

3.6 Implementation of Suffix Tree Algorithms

3.6.1 The Lazy Suffix Tree Algorithm

For the functional implementation of the lazy algorithm, we represent a suffix s of x in constant space by the pair $(s, |s|)$ of type $(subword\ \alpha)$. The function $grouplcp$ (see Definition 3.5.1) is always applied to a non-empty list of suffixes ordered by their length. If this list contains only one element $(s, slen)$, then the length of the longest common prefix is $slen$, and the list $[([\,], 0)]$ of rest suffixes represents the empty string. If all suffixes represented in the list begin with the character a, then a is dropped from these, and $grouplcp$ is applied to the resulting list. Otherwise, the longest common prefix has length 0. Note that the pattern $(c : u, i+1)$ in the list generators only matches non-empty strings. Thus, $grouplcp$ eliminates superfluous strings which result from nested suffixes.

```
grouplcp::[subword α]→(num,[subword α])
grouplcp [(s,slen)] = (slen,[([],0)])
grouplcp ((a:s,slen+1):ss)
  = (prefixlen+1,rests),  if [0 | (c:u,i+1)←ss; a ~= c] = []
  = (0,(a:s,slen+1):ss),  otherwise
    where (prefixlen,rests) = grouplcp ((s,slen):[(u,i) | (c:u,i+1)←ss])
```

The length of a suffix is used only when *grouplcp* is applied to a unitary list. This situation occurs when a label for a leaf edge is to be computed. In [GK95a], the length component of a suffix was omitted and computed on demand in $\mathcal{O}(|s|)$. This led to a considerable simplification and improved the speed by a factor 2. However, the running time was measured *without* evaluating the length of a suffix. The argument was that applications using the length read the characters of the suffix anyway which amortizes the cost of calculating the length. In the thesis, some operations on compact suffix trees occur that do evaluate the length *without* reading the suffix. Therefore, we prefer the more complicated solution that allows to obtain the length of s in constant time, rather than in $\mathcal{O}(|s|)$.

The implementation of the function *group* is straightforward. Let *characters* be the ordered list of characters in $\mathcal{A}$. For each a in *characters*, the suffixes beginning with a are selected and a is dropped from these. If the resulting list is not empty, then it matches the expression $(y : ys)$ and the function *grouplcp* is applied to it. This yields a pair $(j, rests)$. j is the length of the longest common prefix w of the suffixes represented by $(y : ys)$, and *rests* is the list of the remaining suffixes. Taking the first component s of the pair y, we obtain a subword $(a : s, j + 1)$ of x representing aw.

```
group::[α]→[subword α]→[(subword α,[subword α])]
group characters ss = [((a:fst y,j+1),rests) | a←characters; y:ys←[select a];
                                               (j,rests)←[grouplcp (y:ys)]]
                      where select a = [(u,i) | (c:u,i+1)←ss; a = c]
```

The lazy algorithm is implemented by a function *lazytree* which calls a function *subtree* to construct a subtree from a list of suffixes. To obtain the entire suffix tree, *subtree* is applied to all suffixes of x paired with their lengths. If the list of suffixes becomes unitary, *subtree* returns a leaf. Otherwise, it constructs a node with some outgoing edges whose labels are obtained by grouping the suffixes. The outgoing edges lead to subtrees recursively constructed from the corresponding set of remaining suffixes. Note that *lazytree* requires that the length of x and the characters in $\mathcal{A}$ are known in advance.

```
lazytree::[α]→(subword α)→ctree α β
lazytree characters (x,n)
  = subtree (zip2 (suffixes x) [n,n-1..])
    where subtree [([],0)] = N [] undef undef
          subtree ss
            = N es undef undef
              where es = [(aw,subtree rests) | (aw,rests)←group characters ss]
```

Let $l = |\mathcal{A}|$. Since *lazytree* uses iteration over l characters to group suffixes, each character is inspected l times. This leads to an extra factor l. The running time of *lazytree* is therefore

$\mathcal{O}(l \cdot n^2)$ in the worst case, and $\mathcal{O}(l \cdot n \cdot \log n)$ in the average case. The space consumption is $\mathcal{O}(n)$. Thus, *lazytree* is not optimal. The iteration over the characters in $\mathcal{A}$ and the resulting factor can be avoided if one uses counting sort. However, this only works with extension 2. We will not give such an implementation.

Note that *lazytree* constructs compact suffix trees with undefined suffix links and annotations. In section 3.7, we show how to compute suffix links and calculate the annotations, respectively.

3.6.2 Tree Transformers

The linear time suffix tree algorithms build compact suffix trees via local updates at some nodes. Moreover, they construct suffix links in order to have efficient access to these nodes. In this way, the algorithms produce a cyclic graph structure with multiple pointers to nodes. Such a graph structure can be implemented by the recursive type $(ctree\ \alpha\ \beta)$ (see section 3.4.5). Multiple pointers are represented by shared subexpressions. However, in an expression of type $(ctree\ \alpha\ \beta)$, a local update can only be accomplished by its global reconstruction. This would contradict the linear time constraint. Therefore, to implement McCreight's and Ukkonen's suffix tree algorithms purely functionally, we use another representation of the suffix tree. In order to update a tree in-place, the nodes are stored in an updatable array such that each node can only be accessed via its unique index in the array. Thus, our implementation requires extension 2, as defined in section 2.4. To ensure single threadedness, the array will be manipulated by tree transformers which are a special instance of state transformers. We introduce the following type for nodes.

```
node ::= Index num | Leaf | Undefnode
```

The argument of the constructor *Index* is an index into the array. For the linear time suffix tree algorithms, we do not store any information at the leaves. Hence, these are represented by a constructor *Leaf* without index argument. The constructor *Undefnode* stands for the symbol $\perp$ introduced in Definition 3.2.11.

For each node represented by an expression $(Index\ i)$, we store at index i of the array a triple $(es, link, tag)$ of type $([(\alpha, node)], node, \beta)$ where es is the list of the labeled outgoing edges, $link$ is the suffix link, and tag the annotation. Thus, a node is not represented explicitly as a pointer to a subtree. Instead, we use an index i referring to the array. Note that es, $link$, and tag correspond to the arguments of the constructor N which was introduced for the recursive type $(tree\ \alpha\ \beta)$ (cf. Definition 3.4.4). However, as we access inner nodes via indices into an array, we get a non-recursive type, and therefore do not need a new type constructor. We can simply combine es, $link$, and tag to a triple. To select one of the three items from the triple, we introduce the polymorphic functions *first*, *second* and *third*. These will also be used in other contexts.

```
first::(α,β,γ)→α
first (left,mid,right) = left

second::(α,β,γ)→β
second (left,mid,right) = mid
```

```
third::(α,β,γ)→γ
third (left,mid,right) = right
```

In each step of the construction, we must know which of the entries in the array represent the actual $\mathcal{A}^+$-tree, and which are free to store the inner nodes created later. We therefore introduce a *free*-index that divides the array into two parts:

- The entries with the indices $0, \ldots, free - 1$ represent the actual tree.
- The entries with the indices $\geq free$ can be used to store the nodes created later.

At the beginning of the construction the *root* is the only node. Consequently, we store it at index 0 which is expressed as follows:

```
root::node
root = Index 0
```

Example 3.6.1 The atomic $\mathcal{A}^+$-tree in Figure 3.1 has five nodes. It is represented by an array as follows:

```
atreeArray::array ([(char,node)],node,num)
atreeArray = makearray 5 [node0,node1,node2,node3,node4]
             where node0 = ([('a',Index 1),('b',Index 4)],undef,undef)
                   node1 = ([('b',Index 2)],Index 0,undef)
                   node2 = ([('b',Index 3)],Index 4,undef)
                   node3 = ([],Index 4,undef)
                   node4 = ([],Index 0,undef)
```

□

As we always have to provide access to the *free*-index, it becomes part of the state. Hence, the monad *treetrans* of tree transformers is defined as follows:

```
treetrans α β γ == statetrans (state α β) γ
state α β == (array ([(α,node)],node,β),num)
```

The functions *unit* and *bind* are inherited from the monad of state transformers as defined in section 2.3.1. To manipulate the array-component of the state, we follow section 2.3.2 and introduce the functions *block*, *fetch*, and *assign*.

```
block::num→(treetrans α β γ)→γ
block n treetrans = value
                    where (value,state) = treetrans (makearray n vlist,1)
                          vlist = take n (repeat ([],undef,undef))

fetch::node→treetrans α β ([(α,node)],node,β)
fetch (Index i) (array,free) = (lookup i array,(array,free))
```

```
assign::node→([(α,node)],node,β)→treetrans α β ()
assign (Index i) v (array,free) = ((),(update i v array,free))
```

block needs $\mathcal{O}(n)$ time to initialize n entries of the array with the triple $([\,], undef, undef)$. This means that initially every node has an empty list of outgoing edges, an undefined suffix link, and and undefined annotation. Moreover, the *free*-index is initialized to 1. (*fetch node state*) returns in constant time the items stored for *node*. It propagates the *state* unchanged. To avoid duplicating the state, *fetch* must be implemented as follows. First the entry i in the array is fetched and then pair consisting of this value and the unchanged state is returned. (*assign node v state*) updates the array-component of *state* such that the entry with the index specified by *node* contains value v. This takes constant time.

newnode is the only operation that changes the right component of the state, that is, the *free*-index. It returns a new node with index *free* and increments *free* by 1. After calling *newnode*, the entry with the index *free* can be used to store items for the next node to be created. Notice that *newnode* does not access or change the array-component of the state.

```
newnode::treetrans α β node
newnode (array,free) = (Index free,(array,free+1))
```

For convenience we introduce the function *use* which applies a function f to the value returned by a tree transformer *treetrans*. *use* corresponds to the function *using* introduced by Hutton [Hut92] in the context of parser combinators.

```
use::(treetrans α β γ)→(γ→δ)→treetrans α β δ
(treetrans $use f) state = (f value,state')
                           where (value,state') = treetrans state
```

(*seledge ok node*) returns the list $[(w, node')]$ if there is an edge $node \xrightarrow{w} node'$ such that (*ok w*) is true. Otherwise, it returns the empty list. The running time is proportional to the number of edges outgoing from *node*.

```
seledge::(α→bool)→node→treetrans α β [(α,node)]
seledge ok node = fetch node $use (selok.first)
                  where selok es = [(w,node') | (w,node')←es; ok w]
```

(*sellink node*) selects in constant time the suffix link of *node*.

```
sellink::node→treetrans α β node
sellink node = fetch node $use second
```

(*setlink node link*) sets the suffix link of *node* to *link* and returns *link*. This takes constant time.

```
setlink::node→node→treetrans α β node
setlink node link = (fetch node $bind set) $use const link
                    where set (es,undeflink,tag) = assign node (es,link,tag)
```

Since the access to the array is accomplished by *fetch* and *assign* only, all the functions of this section are single-threaded. To preserve this property, we make the type (*treetrans* α β γ) into an abstract data type supporting all functions of this section.

3.6.3 Compact Tree Transformers

Compact tree transformers are special instances of tree transformers.

```
ctreetrans α β γ == treetrans (subword α) β γ
```

Let es be an ordered list of edge labels paired with nodes. Suppose $bpair = ((bv, k), bnode)$. $(insertedges\ es\ bpair)$ inserts $bpair$ into es, yielding an ordered list. If there is a pair $((cu, j), cnode)$ in es such that $c = b$, then it is replaced by $bpair$. The running time of *insertedges* is proportional to the length of es.

```
insertedges::[(subword α,node)]→(subword α,node)→[(subword α,node)]
insertedges es ((b:v,k),bnode)
  = takewhile ((<b).sel) es ++ ((b:v,k),bnode):dropwhile ((<=b).sel) es
    where sel ((c:u,j),cnode) = c
```

$(addleaf\ ay\ node)$ returns $node$ and adds a leaf edge labeled ay to the $node$. Note that we do not store any information for the leaves: A leaf is represented by the constructor *Leaf*.

```
addleaf::(subword α)→node→ctreetrans α β node
addleaf ay node = (fetch node $bind add) $use const node
                  where add (es,link,tag) = assign node (es',link,tag)
                                            where es' = insertedges es (ay,Leaf)
```

$(splitedge\ node\ v\ w\ node')$ splits $node \xrightarrow{vw} node'$ into the edges $node \xrightarrow{v} node'' \xrightarrow{w} node'$. It returns a new node denoted by $node''$. The splitting is accomplished in two steps as follows. At the entry specified by $node''$ the triple $([(w, node')], undef, undef)$ is stored. That is, a new node $node''$ is introduced with a new edge $node'' \xrightarrow{w} node'$. Then the edge $node \xrightarrow{vw} node'$ is replaced by the edge $node \xrightarrow{v} node''$. The running time is proportional to the number of edges outgoing from $node$.

```
splitedge::node→(subword α)→(subword α)→node→(ctreetrans α β node)
splitedge node v w node'
  = fetch node $bind split
    where split (es,link,tag)
            = newnode $bind add
              where add node''
                      = assign node'' ([(w,node')],undef,undef) $bind const replace
                        where replace = assign node (es',link,tag) $use const node''
                              es' = insertedges es (v,node'')
```

The function $(cseledge\ a)$ selects an a-edge.

```
cseledge::α→node→ctreetrans α β [(subword α,node)]
cseledge a = seledge ok
             where ok (c:u,i) = (a = c)
```

3.6.4 Locations

In analogy to section 3.4.6 we introduce two constructors, *LocN* for node locations and *LocE* for edge locations.

```
location α ::= LocN node | LocE node (subword α) (subword α) node
```

For convenience we introduce a function *loc2node* that extracts the node from a node location.

```
loc2node::(location α)→node
loc2node (LocN node) = node
```

The monadic implementation of *getloc* is similar to the implementation given in section 3.4.6. However, it must also cover the case for edge locations, because in the first equation defining *ukkstep* (cf. Definition 3.5.11) *loc* may be an edge location. Let $sz \in words(T)$. Then $(getloc\ (loc_T(s))\ z)$ returns $loc_T(sz)$ and propagates the state unchanged. This takes $\mathcal{O}(l \cdot |z|)$ steps, where l is the average number of edges outgoing from the nodes visited by *getloc*.

```
getloc::(location α)→(subword α)→ctreetrans α β (location α)
getloc loc (z,0) = unit loc
getloc (LocN node) (a:z,j)
  = (cseledge a node $use hd) $bind down
    where down ((v,i),node')
            = unit (LocE node (v,j) w node'),          if node' = Leaf ∨ j < i
            = getloc (LocN node') (advance i (a:z,j)), otherwise
              where w = advance j (v,i)
getloc (LocE node (v,k) (w,i) node') (z,j)
  = unit (LocE node (v,k+j) (advance j (w,i)) node'), if node' = Leaf ∨ j < i
  = getloc (LocN node') (advance i (z,j)),            otherwise
```

Note the differences and similarities between the monadic implementation above and the implementation of *getloc* given in section 3.4.6. In both implementations, the $\mathcal{A}^+$-tree T is not an explicit argument. In the monadic implementation, the information for a certain node is fetched from the array, whereas in the other implementation pattern matching on the constructor N is used. The like holds for the monadic implementation of *linkloc* below and the implementation given in section 3.4.6.

```
linkloc::(location α)→ctreetrans α β (location α)
linkloc (LocN node) = sellink node $use LocN
linkloc (LocE node v w node')
  = sellink node $bind down
    where down link = getloc (LocN node) (advance 1 v),  if node = root
                    = getloc (LocN link) v,              otherwise
```

For convenience we introduce the functions *nodeloc* and *rootloc*. *nodeloc* checks if a location is a node. *rootloc* checks if a location is the *root*.

```
nodeloc::(location α)→bool
nodeloc (LocN node) = True
nodeloc loc = False

rootloc::(location α)→bool
rootloc (LocN node) = (node = root)
rootloc loc = False
```

The function *insert* corresponds to the operator $[\ \leftarrow\ ]$ expressing insertion into $\mathcal{A}^+$-trees (cf. Definition 3.2.11). More precisely: let $(T', \overline{r}) = T[loc \leftarrow ay]$ and suppose the array represents the compact $\mathcal{A}^+$-tree T. Then $(insert\ loc\ ay)$ returns $\overline{r}$ and updates the array such that the new array represents T'.

```
insert::(location α)→(subword α)→ctreetrans α β node
insert (LocN node) ay = addleaf ay node $use const Undefnode
insert (LocE node v w node') ay = splitedge node v w node' $bind addleaf ay
```

3.6.5 Ukkonen's Online Suffix Tree Algorithm

The monadic implementation of the boolean function *occurs* (see Definition 3.2.7) is straightforward.

```
occurs::(location α)→α→ctreetrans α β bool
occurs (LocN node) a = cseledge a node $use (~=[])
occurs (LocE node v (c:w,j) node') a = unit (a = c)
```

The monadic implementation of *ukkstep* (cf. Definition 3.5.11) is as follows:

```
ukkstep::(subword α)→node→(location α)→ctreetrans α β (location α)
ukkstep ay z loc
  = occurs loc a $bind ukkcases
    where (a:y,i) = ay
          ukkcases occ
            = ukklink undef $bind const (getloc loc (a:y,1)),  if occ
            = (insert loc ay $bind ukklink) $use const loc,    if rootloc loc
            = (insert loc ay $bind ukklink) $bind nextstep,    otherwise
              where ukklink r = unit r,                   if z = Undefnode
                              = setlink z (loc2node loc), if occ ∨ r = Undefnode
                              = setlink z r,              otherwise
          nextstep r = linkloc loc $bind ukkstep ay r
```

The implementation closely resembles the definition of *ukkstep*. The identifiers in the implementation are consistent with those in the definition. Of course, r and z stand for $\overline{r}$ and $\overline{z}$, respectively. The case distinction in the definition of *ukkstep* corresponds to the three cases of the function *ukkcases* above. The function *ukklink* is used for setting the suffix links. It is supplied with the argument r, which is either undefined, set to *Undefnode* or to the node just created. The case distinction in *ukklink* reflects the cases defining the new set

L' of suffix links (cf. Definition 3.5.11). Note that it is necessary to set a suffix link if and only if $z \neq$ *Undefnode*. *ukkstep* begins by evaluating the expression (*occurs loc a*). Let *occ* be the value of this expression. Then the following cases occur.

1. If *occ* is true, then the suffix link for z is set to (*loc*2*node loc*) if necessary. Moreover, the location of the next active suffix is computed.
2. If *occ* is false and *loc* is the *root*, then *ay* is inserted at the *root* and the suffix link for z is set to (*loc*2*node loc*) if necessary.
3. If *occ* is false and *loc* is not the *root*, then *ay* is inserted at location *loc*. This yields a value r returned by the function *insert*. Moreover, the suffix link for z is set to (*loc*2*node loc*) or to r if necessary. The *next step* is to compute the location (*linkloc loc*) of the next suffix and to call *ukkstep* with the proper arguments.

The monadic implementation of the function *ukk* (see section 3.5.2) is now straightforward.

```
ukk::(subword α)→(location α)→ctreetrans α β ()
ukk ([],0) loc = unit ()
ukk ay loc = ukkstep ay Undefnode loc $bind ukk (advance 1 ay)
```

We assume a function *showctree* that returns a readable representation of a compact $\mathcal{A}^+$-tree. *callukk* calls *block*. This initializes the state. Moreover, *ukk* is called, followed by *showctree*. This yields a readable representation of the constructed tree which is returned after deallocating the state. Recall that we only store information for the inner nodes of the compact suffix tree. Therefore, it suffices to create an array of size $n = |x|$.

```
callukk::(ctreetrans α β γ)→[α]→γ
callukk showctree x = block n (ukk (x,n) (LocN root) $bind const showctree)
                      where n = #x
```

From the above it is clear that *callukk* takes $\mathcal{O}(|\mathcal{A}| \cdot n)$ time and requires $\mathcal{O}(n)$ space. Hence, the monadic implementation is optimal.

3.6.6 McCreight's Suffix Tree Algorithm

It is easily verified that in *mccstep* (cf. Definition 3.5.19) the first argument of the function *scanprefix* is always a node location. Moreover, the second argument of *scanprefix* is a non-empty suffix of x, due to the presence of the sentinel character in x. Finally, we do not need to compute the list of nodes visited during a traversal, as done by the function *scanprefix*$'$. Hence, the monadic implementation of *scanprefix* is much simpler than the implementation given in section 3.4.6. It can easily be developed from the function *scanprefix*$'$ by translating its second equation to monadic style.

```
scanprefix::node→(subword α)→ctreetrans α β (location α,subword α)
scanprefix node (a:w,j+1)
  = cseledge a node $bind down
    where down [] = unit (LocN node,(a:w,j+1))
          down (((c:u,i+1),node'):es)
            = unit (loc',advance k (w,j)),          if node' = Leaf ∨ k < i
            = scanprefix node' (advance k (w,j)),   otherwise
              where k = lcp u w,                    if node' = Leaf
                      = lcp (take i u) w, otherwise
                    loc' = LocE node (c:u,1+k) (advance k (u,i)) node'
```

The monadic implementation of *mccstep* (cf. Definition 3.5.19) is as follows:

```
mccstep::(location α,subword α,node)→ctreetrans α β (location α,subword α,node)
mccstep (loc,cy,z)
  = linkloc loc $bind mcccases
    where
    mcccases loc'
      = scanprefix root (advance 1 cy) $bind justinsert,       if rootloc loc
      = case2 (loc2node loc'),                                 if nodeloc loc
      = setlink z (loc2node loc') $bind case2,                 if nodeloc loc'
      = (insert loc' cy $bind setlink z) $use triple loc' cy,  otherwise
        where case2 node = scanprefix node cy $bind justinsert
              justinsert (loc'',y') = insert loc'' y' $use triple loc'' y'


triple::α→β→γ→(α,β,γ)
triple left mid right = (left,mid,right)
```

The implementation closely resembles the definition of *mccstep*. The identifiers are consistent with those of the definition. r and z refer to $\overline{r}$ and $\overline{z}$, respectively. The case distinction in the definition of *mccstep* corresponds to the four cases of the function *mcccases* above. *mccstep* applies *linkloc* to obtain the location loc' of the next suffix. Then it proceeds as follows:

1. If loc is the *root*, then the pair (loc'', y') is computed by applying *scanprefix* to the *root* and the string y which is obtained by advancing one character in cy. Then y' is inserted at loc'' which yields a value r that is returned together with loc'' and y'.

2. If loc is a node, then so is loc'. If furthermore loc is not the *root*, then (loc'', y') is computed by applying *scanprefix* to $(loc2node\ loc')$ and the string cy. Then y' is inserted at loc'', which yields a node r that is returned together with loc'' and y'.

3. If loc' is a node but not loc, then the suffix link for z is set to $(loc2node\ loc')$. The rest is like in case 2.

4. If loc' is not a node, then $(loc'', y') = (loc', cy)$. Hence, cy is inserted at loc', yielding a new inner node r. The suffix link for z is set to r, and r together with loc' and cy is returned.

Note that the value r does not explicitly occur in the implementation of *mccstep*. After r is returned by the function *insert*, it is propagated by *bind* to serve as the second argument of *setlink* which itself returns r, so that it can be used as the third argument for the function *triple*.

The monadic implementation of the function *mcc* is now straightforward.

```
mcc::(location α,subword α,node)→ctreetrans α β ()
mcc (loc,(a:y,i),z)
  = unit (),                              if y = [] & rootloc loc
  = mccstep (loc,(a:y,i),z) $bind mcc,    otherwise
```

callmcc calls *block*. This initializes the state. Moreover, a leaf edge labeled x is added to the *root*. Then *mcc* is called with the proper initial arguments. Finally, a readable representation of the constructed tree is obtained by *showctree*. The representation is returned after deallocating the state.

```
callmcc::(ctreetrans α β γ)→[α]→γ
callmcc showctree x
  = block n ((addedge $bind mcc) $bind const showctree)
    where addedge = addleaf (x,n) root $use const (LocN root,(x,n),undef)
          n = #x
```

From the above it is clear that *callmcc* takes $\mathcal{O}(|\mathcal{A}| \cdot n)$ time and $\mathcal{O}(n)$ space. Hence, the implementation is optimal.

The implementations of *ukkstep* and *mccstep* are tailored so that they closely resemble Definitions 3.5.11 and 3.5.19, respectively. The development was not straightforward. It takes some time to get used to the monadic programming style. The main problem is that in order to guarantee the single-threadedness, the compact tree transformers do not explicitly have the compact $\mathcal{A}^+$-tree as an argument. This means that one has to do without an important technique for structuring functional programs. The intermediate results, namely the created and visited nodes cannot be bound to identifiers using a *where* clause. Instead, they are implicitly passed between the compact tree transformers or they occur as arguments to some locally defined functions. This often leads to programs that are relatively hard to understand. In a functional language which allows λ-expressions (for instance Haskell or Clean) this notational disadvantage can be relaxed: Instead of introducing local functions, one has anonymous functions which are introduced inside ordinary expressions. For an example see [Wad92b].

3.7 Computing Suffix Links and Annotations

Suffix links are very important in many applications of $\mathcal{A}^+$-trees. In atomic $\mathcal{A}^+$-trees, for instance, the suffix links allow to find all occurrences of some patterns in an input string in linear time (see section 5.2). In compact suffix trees the suffix links are used for determining the q-gram distance (see section 3.9.3), for sublinear exact and approximate string searching (see sections 4.4, 4.5, 6.8), and for computing matching statistics (see section 6.6). This section shows how to compute suffix links efficiently.

3.7.1 Suffix Links for Atomic $\mathcal{A}^+$-Trees

Let T be an atomic $\mathcal{A}^+$-tree. To construct the suffix links of T, we use the algorithm of Aho and Corasick for computing the failure transitions (see [AC75, Algorithm 3]). The idea of this algorithm can be described as follows: Suppose that $\overline{sa} \longrightarrow \overline{v}$ is a suffix link in T. The node $\overline{v}$ is determined as follows. If $\overline{s} = root$, then $\overline{v} = root$. If $\overline{s} \neq root$, then let $\overline{s} \longrightarrow \overline{u} \in links(T)$. Since u is a suffix of s, we can conclude $sa \sqsupset ua$. Assume that v is longer than ua. Then $v = wa$ and therefore w is a suffix of s such that $\overline{w} \in nodes(T)$. This is a contradiction since u is the longest suffix of s such that $\overline{u} \in nodes(T)$. Hence, v is the longest suffix of ua such that $\overline{v} \in nodes(T)$. That is, $\overline{v}$ can be found by following the suffix link path starting at $\overline{u}$, until a node with an a-edge is found or the $root$ is reached. This is accomplished by the function $next$.

Definition 3.7.1 The function $next : nodes(T) \times \mathcal{A} \to nodes(T)$ is defined as follows:

$$next(\overline{s}, a) = \begin{cases} \overline{sa}, & \text{if } \overline{s} \xrightarrow{a} \overline{sa} \in edges(T) \\ root, & \text{else if } \overline{s} = root \\ next(\overline{v}, a), & \text{otherwise} \\ & \text{where } \overline{s} \longrightarrow \overline{v} \in links(T) \end{cases}$$

Lemma 3.7.2 Let $\overline{u}$ be a node in T and $\overline{v} = next(\overline{u}, a)$ for some $a \in \mathcal{A}$. Then v is the longest suffix of ua such that $\overline{v}$ is a node in T.

Proof Routine. □

The implementation of the function $next$ is straightforward.

```
next::(tree α β)→α→(tree α β)
next (N es link tag) a = hd asucc,      if asucc ~= []
                       = N es link tag, if link = Undeftree
                       = next link a,   otherwise
                         where asucc = [node | (c,node)←es; a = c]
```

For the sake of good readability and practical performance, we use list comprehensions to select an a-edge, thereby ignoring the fact that the list of outgoing edges is ordered. The function *addlinks* computes the suffix links of T in a single tree traversal. The main work is done by the function *setlink* which sets the suffix link for $\overline{s}$ to the node $\overline{u}$ as specified above. More precisely, if $\overline{s}$ is represented by the expression $(N\ es\ link\ tag)$ and $\overline{u}$ is represented by $link'$ then $link$ is replaced by $link'$. Furthermore, for each edge $\overline{s} \xrightarrow{a} \overline{sa}$ outgoing from $\overline{s}$, *setlink* is called with the arguments $\overline{sa} = node$ and $\overline{v} = next(link', a)$. This yields an a-edge leading to a subtree in which the suffix links are set.

```
addlinks::((tree α β)→(tree α β)→(tree α β))→(tree α β)→tree α β
addlinks setlink (N es link tag)
  = root where root = N [(a,setlink node root) | (a,node)←es] Undeftree tag

setlink::(tree α β)→(tree α β)→(tree α β)
setlink (N es link tag) link'
  = N es' link' tag
    where es' = [(a,setlink node (next link' a)) | (a,node)←es]
```

We have implemented *addlinks* as a higher order functions since it will be supplied with a variation of *setlink* that performs some additional calculations (see section 5.2). According to the description above, the suffix link $\overline{s} \longrightarrow \overline{u}$ must be computed *before* the suffix link $\overline{sa} \longrightarrow \overline{v}$. Because of the laziness, we do not have to worry about this order: suffix links are computed when they are demanded. In an eager language the solution is not so straightforward. Like in Aho and Corasick's imperative implementation, one can, for instance, use a queue to implement a breadth first traversal. This guarantees that the computations are done in the right order. However, it leads to a much longer and more complicated program.

Theorem 3.7.3 *addlinks* computes the suffix links of T in $\mathcal{O}(l \cdot |T|)$ steps where l is the average number of edges outgoing from the nodes visited by *next*.

Proof The correctness of the implementation is clear. To analyze the efficiency of *addlinks*, we have to determine the number of calls to the function *next*. First observe that in each call of the form $(next\ \overline{s}\ a)$ either

(1) a different edge label a is consumed (first and second equation implementing *next*), or

(2) a suffix link $\overline{s} \longrightarrow \overline{v} \in links(T)$ is traversed (third equation implementing *next*).

The number of steps of type (1) is bound by $|edges(T)|$. Moreover, for each step of type (2) there are $|s| - |v| > 0$ steps of type (1). Thus, the number of steps of type (2) is bound by $|edges(T)|$, too. Therefore, there are at most $(2 \cdot |edges(T)|)$ calls to the function *next*. □

Note that Aho and Corasick [AC75] did not give a proof of the above efficiency result and we have not seen such a proof elsewhere.

3.7.2 Suffix Links for Compact Suffix Trees

To our knowledge there is no application[3] of compact suffix trees that requires the suffix links of the leaves to be set. This is due to the fact that these are in general not atomic which means that they cannot be used to provide efficient access from the location of a string cy to the location of y (cf. section 3.2.2). Therefore, we restrict ourselves to the problem of computing the inner suffix links of a compact suffix tree efficiently. Of course, this problem does not occur when the tree is constructed by Ukkonen's or McCreight's algorithm since these construct the inner suffix links anyway. The problem is motivated by the fact that the lazy algorithm does not compute any suffix links. Giegerich and Kurtz [GK95a] have already given a solution. We adopt this. However, while in [GK95a] only the code of a function is given, we also present the main idea which is not trivial. Moreover, we additionally prove the linearity claim of [GK95a] and show how to improve the solution of Giegerich and Kurtz.

In this section, let T be the compact suffix tree for some string x of length n. The idea of computing the inner suffix links of T is similar to Aho and Corasick's algorithm for computing failure transitions. It can be described as follows:

Suppose $\overline{cy} \in inner(T)$. Then $\overline{y} \in inner(T)$ as well. Hence, $loc_T(y) = \overline{y}$. Let $\overline{u} \xrightarrow{av} \overline{cy}$ be an edge in T. Then $uav = cy$. $\overline{y}$ is determined as follows:

[3] Except for building the compact "affix tree" of x by reverse unification of $cst(x)$ and $cst(x^{-1})$. See [Sto95].

1. If $\overline{u} = root$, then $av = cy$, i.e., $v = y$. Hence, $\overline{y} = loc_T(y) = loc_T(v) = getloc(root, v)$.

2. If $\overline{u} \neq root$, then $\overline{u} \longrightarrow \overline{z} \in innerlinks(T)$ and $u = bz$ for some $b \in \mathcal{A}$. Therefore, we get $bzav = uav = cy$ which implies $zav = y$. Hence, $\overline{y} = loc_T(y) = loc_T(zav) = getloc(loc_T(z), av) = getloc(\overline{z}, av)$.

According to this description, the function *cstlinks* (as shown below) computes the inner suffix links of a compact suffix tree, leaving the suffix links for the leaves undefined. It does so in a single tree traversal. The second case above implies that the suffix link $\overline{u} \longrightarrow \overline{z}$ must be computed before the suffix link $\overline{cy} \longrightarrow \overline{y}$. Due to the laziness, we do not have to worry about this order (cf. section 3.7.1).

```
cstlinks::(ctree α β)→(ctree α β)
cstlinks (N es link tag)
  = root
    where root = N es' Undeftree tag
          es' = [(av,setcstlink node (getloc root (advance 1 av))) | (av,node)←es]


setcstlink::(ctree α β)→(location α β)→(ctree α β)
setcstlink (N [] link tag) loclink = N [] undef tag
setcstlink (N es link tag) (LocN link')
  = N es' link' tag
    where es' = [(w,setcstlink node (getloc link' w)) | (w,node)←es]
```

cstlinks is very similar to the function *addlinks* from section 3.7.1. The functions *getloc* and *setcstlink* play the role of the function *next* and *setlink* in *addlinks*. The main difference is that *cstlinks* sets the suffix links for the leaves to *undef*. Moreover, the suffix links for the nodes immediately below the *root* are in general not set to the *root*.

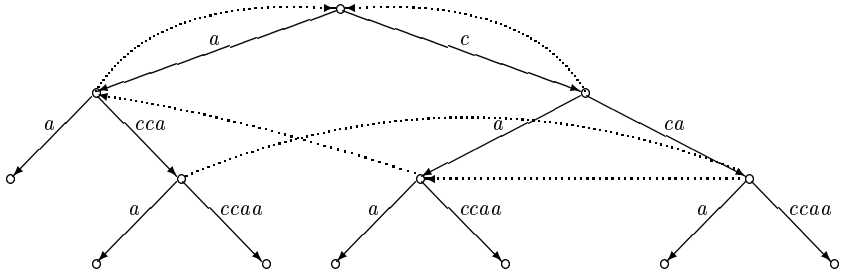
To determine the efficiency of *cstlinks*, one first notes that there are $\mathcal{O}(n)$ inner nodes in T, and therefore $\mathcal{O}(n)$ suffix links to be set. Hence, there are $\mathcal{O}(n)$ calls to the function *setcstlink*. The main part of the work is done by the function *getloc*. There are $\mathcal{O}(n)$ calls of the form $(getloc\ \overline{u}\ \varepsilon)$, each taking constant time. Since each call of the form $(getloc\ \overline{u}\ w)$, $w \neq \varepsilon$, consumes a non-empty prefix of w, the number of these calls is bound by the length of the labels of the inner edges. To state this precisely, we define a function $elen : edges(T) \to \mathbb{N}_0$:

$$elen(\overline{s} \xrightarrow{av} \overline{sav}) = \begin{cases} 0, & \text{if } \overline{sav} \text{ is a leaf} \\ |v|, & \text{else if } \overline{s} = root \\ |av|, & \text{otherwise} \end{cases}$$

The number of calls of the form $(getloc\ \overline{u}\ w)$, $w \neq \varepsilon$ is at most $\sum_{e \in edges(T)} elen(e)$. Unfortunately, we do not know whether this upper bound is in general proportional to n. So we cannot prove the linearity this way and present a second approach which is indeed successful. First note that for each call of the form $(getloc\ \overline{u}\ vz)$, $vz \neq \varepsilon$, the following is evident:

- It traverses an edge $u \xrightarrow{v} \overline{uv}$.
- It occurs if and only if some suffix link $\overline{auvz} \longrightarrow \overline{uvz}$ is to be computed.

Figure 3.7: $cst(accaccaa)$ with the Inner Links



Example 3.7.4 When *cstlinks* computes the inner suffix links for $cst(accaccaa)$ (see Figure 3.7), we observe the following:

- $(getloc\ root\ a)$ traverses the edge $root \xrightarrow{a} \overline{a}$ to compute the suffix link $\overline{ca} \longrightarrow \overline{a}$.
- $(getloc\ root\ cca)$ traverses the edge $root \xrightarrow{c} \overline{c}$ to compute the suffix link $\overline{acca} \longrightarrow \overline{cca}$.
- $(getloc\ \overline{c}\ ca)$ traverses the edge $\overline{c} \xrightarrow{ca} \overline{cca}$ to compute the suffix link $\overline{acca} \longrightarrow \overline{cca}$.
- $(getloc\ root\ ca)$ traverses the edge $root \xrightarrow{c} \overline{c}$ to compute the suffix link $\overline{cca} \longrightarrow \overline{ca}$.
- $(getloc\ \overline{c}\ a)$ traverses the edge $\overline{c} \xrightarrow{a} \overline{ca}$ to compute the suffix link $\overline{cca} \longrightarrow \overline{ca}$. □

Lemma 3.7.5 *getloc* traverses each edge in T at most $|\mathcal{A}|$ times.

Proof Suppose $\overline{u} \xrightarrow{v} \overline{uv}$ is an edge in T which is traversed by *getloc*. Obviously, $\overline{u}$ and $\overline{uv}$ are inner nodes. Moreover, there exists a character a and a string z such that $\overline{auvz}$ is an inner node in T and the following statements hold:

- $u = \varepsilon$ and either $root \xrightarrow{avz} \overline{avz} \in edges(T)$ or $\overline{a} \xrightarrow{vz} \overline{avz} \in edges(T)$.
- $u \neq \varepsilon$, $u = wy$ for some strings w, y such that $\overline{aw} \xrightarrow{yvz} \overline{auvz} \in edges(T)$.

To show the uniqueness of z, suppose there is a string z' such that $\overline{avz'}$ is an inner node in T. Consider the following cases:

- $u = \varepsilon$. If $root \xrightarrow{avz'} \overline{avz'} \in edges(T)$, then $\overline{a}$ is not a node in T. Therefore, we get $root \xrightarrow{avz} \overline{avz} \in edges(T)$ which implies $z = z'$. If $\overline{a} \xrightarrow{vz'} \overline{avz'} \in edges(T)$, then $root \xrightarrow{a} \overline{a} \in edges(T)$. Hence, there is no edge $root \xrightarrow{avz} \overline{avz}$ from which we conclude $\overline{a} \xrightarrow{vz} \overline{avz} \in edges(T)$. This implies $z = z'$.
- $u \neq \varepsilon$. Suppose $u = w'y'$ for some strings w', y' such that $\overline{aw'} \xrightarrow{y'vz'} \overline{auvz'} \in edges(T)$. Without restriction to generality we can assume that w' is a prefix of w. Hence, there is a string s such that $w = w's$ and $sy = y'$. Assume $s \neq \varepsilon$. Then $aw's = aw$ is *not* a right-branching subword of x since we have an edge $\overline{aw'} \xrightarrow{syvz'} \overline{auvz'}$. This is a contradiction. Hence, $s = \varepsilon$ which implies $w = w'$ and $y = y'$. Let c be the first character of yv. Since there is only one c-edge outgoing from $\overline{aw}$, we finally get $yvz' = yvz$ which implies $z = z'$. □

According to Lemma 3.7.5, there are $\mathcal{O}(|\mathcal{A}| \cdot n)$ calls to *getloc* in the worst case. Each call takes $\mathcal{O}(l)$ steps, where l is the average number of outgoing edges of the nodes visited by *getloc*. In the worst case $l = |\mathcal{A}|$. Hence, *cstlinks* computes the inner suffix links for T in $\mathcal{O}(|\mathcal{A}|^2 \cdot n)$ steps in the worst case. Under the assumption that the alphabet $\mathcal{A}$ is constant, this shows the linearity claimed in [GK95a]. However, the factor $|\mathcal{A}|^2$ is unsatisfying. It remains to show a tighter upper bound.

The speed of *cstlinks* can be improved by exploiting the ordering of the edge labels: Let $\overline{cy}$ and $\overline{y}$ be some inner nodes in T with a list es and es'' of outgoing edges, respectively. Let $\overline{cy} \xrightarrow{av} \overline{cyav}$ be an edge in T that leads to an inner node $\overline{cyav}$. Then there is an a-edge in es'' that must be traversed in order to compute the suffix link for $\overline{cyav}$. Selecting this a-edge in a brute force manner takes $\mathcal{O}(|es''|)$ steps on the average. This adds up to $\mathcal{O}(|es| \cdot |es''|)$ steps for all edges in es. Alternatively, we can scan es and es'' simultaneously in $\mathcal{O}(|es''|)$ steps, thereby exploiting the fact that the two lists are ordered.

```
selectfast::[cedge α β]→[cedge α β]→[cedge α β]
selectfast [] es'' = []
selectfast (((a:v,j),node):es) (((c:u,i),node'):es'')
  = ((a:v,j),setcstlink node loclink):selectfast es es'', if a = c
  = selectfast (((a:v,j),node):es) es'',                   otherwise
    where loclink = getloc node' (advance i (a:v,j))
```

The function *selectfast* implements the simultaneous scan of es and es''. If we compute es' in the last line of *setcstlinks* by the expression $(selectfast\ es\ (seledges\ link'))$, then we obtain a faster variant of *cstlinks*.

3.7.3 Annotations for Compact Suffix Trees

The various forms of compact suffix trees essentially differ in the specific annotations of the nodes. Using the polymorphic type $(ctree\ \alpha\ \beta)$, together with a higher order function, we *represent* and *compute* all these forms in a unified way. In particular, we supply a function implementing the lazy suffix tree algorithm with a function that computes an annotation.

Definition 3.7.6 An *annotation function* is a function of type

```
annotationfunction α β == num→[cedge α β]→β
```

Let T be a compact suffix tree. For each S-annotation φ of T considered in the thesis, we provide an annotation function *annotate* such that for each node $\overline{s}$ in T we have $\varphi(\overline{s}) = annotate\ d\ es$ where d is the depth of the node $\overline{s}$ and es is the list of edges outgoing from $\overline{s}$. In other words, if a node is represented by an expression $(N\ es\ link\ tag)$ then tag is computed from the depth of the node and the list es.

The function *cstannotation* applies an annotation function to a compact suffix tree, yielding the tree with its annotation. The suffix links are set to *undef*. Note that it does not make sense to set the suffix link to the expression *link* since this is either undefined or it is an old link which will point to obsolete subtrees after computing the annotation.

```
cstannotation::(annotationfunction α β)→(ctree α γ)→(ctree α β)
cstannotation annotate
  = cstanno' 0
    where cstanno' d (N es link tag)
            = N es' undef (annotate d es')
              where es' = [((w,j),cstanno' (d+j) node) | ((w,j),node)←es]
```

Example 3.7.7 *depth* is a simple annotation function. It computes the depth of a node.

```
depth::annotationfunction α num
depth d es = d
```

If we apply (*cstannotation depth*) to the expression *actree*, defined in Example 3.4.6, we obtain the following tree.

```
actreedepth::ctree char num
actreedepth
  = root
    where root = N [(("abca",4),N [] undef 4),
                    (("bca",3),N [] undef 3),
                    (("ccabca",1),cN)] undef 0
          cN = N [(("abca",4),N [] undef 5), (("cabca",5),N [] undef 6)] undef 1
```

shortestpathlen is an annotation function that computes for each node $\overline{v}$ in T the length of the shortest path from $\overline{v}$ to some leaf in T. If $T = cst(x\$)$, then this information can be used to determine in $\mathcal{O}(|\mathcal{A}| \cdot |w|)$ steps the rightmost position in x where w ends.

```
shortestpathlen::annotationfunction α num
shortestpathlen d es = 0,                                          if es = []
                     = min [i+splen | ((w,i),N es' link splen)←es], otherwise
```

With a similar annotation function one computes the leftmost occurrence of w in x. □

If we have more than one annotation, say $\varphi_1, \ldots, \varphi_k$, we store the tuple $(\varphi_1(\overline{s}), \ldots, \varphi_k(\overline{s}))$ at each node $\overline{s}$ of T. The name of the corresponding annotation function is then obtained by concatenating the names of the k annotations. In chapters 4 and 6, we will see several examples.

3.7.4 Merging the Computations

To construct a compact suffix tree including its inner suffix links and annotations, we can simply compose *lazytree*, *cstlinks*, and *cstannotate* as follows.

```
lazytree'::[α]→(subword α)→(annotationfunction α β)→(ctree α β)
lazytree' characters (x,n) annotate
  = cstlinks (cstannotation annotate (lazytree characters (x,n)))
```

Note that it does not make sense to reverse the order of *cstlinks* and *cstannotation*. This would yield a suffix tree without suffix links since *cstannotation* sets the suffix links to *undef*. Due to the laziness, the construction of the tree, the calculation of the annotations, and the setting of the suffix links are not clearly separated in phases. Instead, if *lazytree'* is applied, the three computations are meshed according to the order in which the values are demanded by other calculations. The efficiency of *lazytree'* can considerably be improved by merging the three computations. This was already suggested in [GK95a].

```
lazytree'':[α]→(subword α)→(annotationfunction α β)→ctree α β
lazytree'' characters (x,n) annotate
  = root
    where root = N es Undeftree (annotate 0 es)
                 where es = [((a:v,j),subtree rests (getloc root (v,j-1)) j) |
                             ((a:v,j),rests)←group characters ss]
                       ss = zip2 (suffixes x) [n,n-1..]
          subtree [([],0)] loclink d = N [] undef (annotate d [])
          subtree ss (LocN link) d
            = N es link (annotate d es)
              where es = [((w,j),subtree rests (getloc link (w,j)) (d+j)) |
                          ((w,j),rests)←group characters ss]
```

lazytree'' is equivalent to *lazytree'*. It can be developed systematically by some program transformation steps. Note that the merging of the phases is not possible for suffix tree constructions that are based on updates (see *naiveOnline* or *naiveInsertion*). As noted in [GK95a], this is due to the following fact. When updating a tree that includes suffix links, the links from elsewhere would still retain their old values and hence point to obsolete subtrees.

For convenience we introduce a function *cst* that uses the lazy algorithm to compute the compact suffix tree for x including the suffix links and annotations. The ordered list representing the different characters in x is obtained by a function *getalpha*. This function can be implemented by sorting all characters in x and removing adjacent duplicates from the resulting list. Using merge sort [CLR90], *getalpha* takes $\mathcal{O}(n \cdot \log n)$ steps.

```
cst::(subword α)→(annotationfunction α β)→ctree α β
cst (x,n) annotate = lazytree'' (getalpha x) (x,n) annotate
```

The function *cst* is an important part of our string processing machinery. In all applications compact suffix trees are constructed in a unified way using *cst*. The decision to take the lazy suffix tree algorithm is justified by practical reasons:

- It perfectly fits into the functional world.
- It is fast on the average.
- It constructs the paths of the suffix tree incrementally as they are traversed, leaving incomplete those subtrees that are never actually needed.

To our opinion these virtues outweigh the disadvantage of non-linear asymptotic running time. Note that for simplicity, in all programs involving suffix tree constructions, we assess the running time of *cst* with $\mathcal{O}(|\mathcal{A}| \cdot n)$. Thus, if we say that a program is optimal, we mean that it is optimal modulo lazy suffix tree construction.

3.8 Deterministic Finite Automata

Automata are mathematical models of devices that process information by giving responses to inputs. The simplest automaton is the deterministic finite automaton (DFA for short). It is a recognition device which has a finite number of states. A state contains the information resulting from the input string read so far. This information is used to determine the reaction of the automaton on the next input character.

In this section, we briefly recall the concept of deterministic finite automata. We are mainly interested in implementation issues. In sections 6.2.4 and 6.5, we give constructions for particular classes of DFAs which can be used for approximate string searching. For a comprehensive overview of (deterministic) finite automata we refer the reader to the survey of Perrin [Per90]. Following [ASU85] we define a DFA as follows:

Definition 3.8.1 A *deterministic finite automaton* is a 4-tuple $(\mathcal{S}, \mathcal{F}, s_0, nextstate)$ where

1. $\mathcal{S}$ is a finite set of *states*,
2. $\mathcal{F} \subseteq \mathcal{S}$ is a distinguished set of *accepting states*,
3. $s_0 \in \mathcal{S}$ is the *initial state*,
4. $nextstate : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a function, the *transition function*.

For each $s \in \mathcal{S}$ and each $b \in \mathcal{A}$, $nextstate(s, b)$ is a transition which yields the *b-successor* of s. □

The DFA $M = (\mathcal{S}, \mathcal{F}, s_0, nextstate)$ begins in state s_0 and reads the characters of an input string one at a time. If the automaton is in state s and reads the character b, it "moves" from state s to the b-successor of s. Whenever the current state is a member of $\mathcal{F}$, M is said to have accepted the string read so far. This is formally defined as follows:

Definition 3.8.2 Let $M = (\mathcal{S}, \mathcal{F}, s_0, nextstate)$ be a DFA and $t \in \mathcal{A}^n$. The *protocol* of t w.r.t. M is the sequence $(s_0, s_1, \ldots, s_n)$ of states such that $s_{j+1} = nextstate(s_j, t_{j+1})$ for each $j, 0 \leq j \leq n-1$. M accepts $t_1 \ldots t_j$ if $s_j \in \mathcal{F}$. □

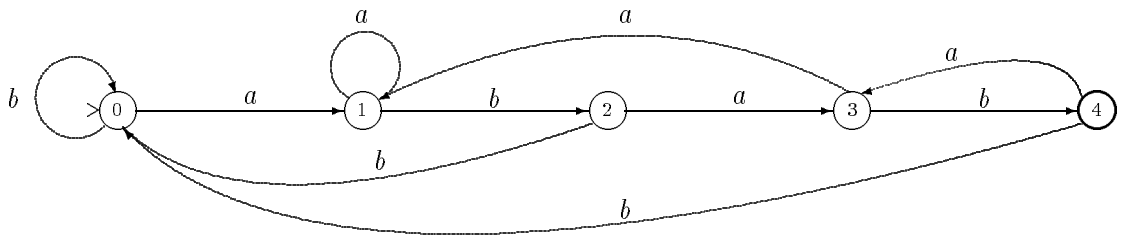
In the terminology of Perrin [Per90], a protocol is called a *path*.

Example 3.8.3 Let $\mathcal{S} = \{0, 1, 2, 3, 4\}$, $\mathcal{F} = \{4\}$, $s_0 = 0$, $\mathcal{A} = \{a, b\}$ and

$$\begin{aligned} &nextstate(0, b) = nextstate(2, b) = nextstate(4, b) = 0 \\ &nextstate(0, a) = nextstate(1, a) = nextstate(3, a) = 1 \\ &nextstate(1, b) = 2 \\ &nextstate(2, a) = nextstate(4, a) = 3 \\ &nextstate(3, b) = 4. \end{aligned}$$

Obviously, $M = (\mathcal{S}, \mathcal{F}, s_0, nextstate)$ is a DFA. A *state transition graph* is a widely used and clear representation of a DFA, albeit only for small examples. For M such a graph is given

Figure 3.8: A State Transition Graph



in Figure 3.8. State 0 (marked by the symbol >) is the initial state, state 4 (shown as a thick circle) is the only accepting state. A directed edge from state i to state j labeled b represents the transition $nextstate(i, b) = j$. The protocol for *aababab* w.r.t. M is 0 1 1 2 3 4 3 4 0. Hence, M accepts *aabab* and *aababab*. One can show that M accepts a string v if and only if *abab* is a suffix of v. Hence, M can be used to find all occurrences of *abab* in some input string over the alphabet $\{a, b\}$. □

3.8.1 Implementation

A DFA can be implemented by an expression of type $(tree\ \alpha\ bool)$: The states correspond to the nodes of the tree. The accepting states are represented by a boolean annotation. A transition $nextstate(s, b) = s'$ corresponds to an edge $\overline{s} \xrightarrow{b} \overline{s'}$. However, this representation leads to a factor $|\mathcal{A}|$ when evaluating a state transition. Since the transition function *nextstate* is a total function, it makes sense to use for each state an $\mathcal{A}$-indexed array, which contains pointers to the successors of s. Using this representation, each state transition can be evaluated in constant time.

Since our extensions (see section 2.4) only provide arrays that are indexed by integers, we presuppose a suitable encoding of $\mathcal{A}$. In particular, we assume that $\mathcal{A}$ is given by a pair $(characters, encode)$ of type

```
alphabet α == ([α],α→num)
```

characters is a list of the form $[c_0, c_1, \ldots, c_{l-1}]$ containing all characters of $\mathcal{A}$ in ascending order, that is, $c_0 < c_1 < \cdots < c_{l-1}$ where $<$ is the built-in ordering. *encode* is a function defined by $(encode\ c_i = i)$. We assume that for each element c_i in $\mathcal{A}$ the expression $(encode\ c_i)$ is evaluated in constant time. Note that the alphabet representation requires $\mathcal{O}(|\mathcal{A}|)$ space.

Example 3.8.4 The nucleotides Adenin, Cytosin, Guanin, and Thymin can be represented by the following type:

```
nucleotide ::= A | C | G | T
```

This implicitly introduces the ordering $A < C < G < T$. The DNA-alphabet can now be represented as follows:

```
dnaAlpha::alphabet nucleotide
dnaAlpha = ([A,C,G,T],encode)
           where encode A = 0
                 encode C = 1
                 encode G = 2
                 encode T = 3
```

□

Definition 3.8.5 Let $M = (\mathcal{S}, \mathcal{F}, s_0, nextstate)$ be a DFA. Each state $s \in \mathcal{S}$ is represented by an expression $(S\ accept\ succ)$ of type

```
dfa ::= S bool (array dfa)
```

such that the following holds:

- *accept* is true if and only if $s \in \mathcal{F}$.
- for each $b \in \mathcal{A}$, the expression $(lookup\ (encode\ b)\ succ)$ returns the representation of the b-successor of s.

M is implemented by the expression representing s_0. □

A DFA-representation is a circular structure, in which certain expressions are shared. Each state and each transition is represented in constant space. Hence, the space requirement is $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{A}|)$. Note that the type *dfa* requires extension 1.

Example 3.8.6 The DFA of Example 3.8.3 is represented by the following expression.

```
dfa1::dfa
dfa1 = s0
       where s0 = S False (makearray 2 [s1,s0])
             s1 = S False (makearray 2 [s1,s2])
             s2 = S False (makearray 2 [s3,s0])
             s3 = S False (makearray 2 [s1,s4])
             s4 = S True  (makearray 2 [s3,s0])
```

□

Note that the type *dfa* is independent of the alphabet. Hence, we can also use *dfa*1 to find all occurrences of the pattern *cdcd* in an input string over the alphabet $\{c, d\}$.

Suppose a DFA $M = (\mathcal{S}, \mathcal{F}, s_0, nextstate)$ is given by an expression $s0$. The function *dfarun* applies M to an input string t and returns a list *jlist* of indices such that j is in *jlist* if and only if M accepts $t_1 \ldots t_j$.

```
dfarun::(alphabet α)→dfa→[α]→[num]
dfarun (characters,encode) s0 t
  = [j | (j,S True succ)←zip2 [0..] (scanl nextstate s0 t)]
    where nextstate (S accept succ) b = lookup (encode b) succ
```

A state transition is evaluated in constant time by the function *nextstate*. The protocol is easily obtained in $\mathcal{O}(|t|)$ steps using the function *scanl* which was defined in section 2.1.4.

3.9 String Comparisons

The comparison of strings is an important operation applied in several fields, such as molecular biology, speech recognition, computer science, and coding theory (see [KS83] for an overview). The most important model for string comparison is the model of edit distance. It measures the distance between strings in terms of edit operations, that is, deletions, insertions, and replacements of single characters. Two strings are compared by determining a sequence of edit operations that converts one string into the other and minimizes the sum of the operations' costs. Such a sequence can be computed in time proportional to the product of the lengths of the two strings, using the technique of dynamic programming.

As stated in [KS83], the edit distance is a measure of local similarities in which matches between subwords are highly dependent on their relative positions in the strings. There are situations where this property is not desired. Suppose one wants to consider strings as similar which differ only by an exchange of large subwords. This occurs, for instance, if a text file has been created from another by a block move operation. In such a case, the edit distance model should not be used since it gives a large string distance. There are two other string comparison models that are more appropriate for this case: The maximal matches model of Ehrenfeucht and Haussler [EH88] and the q-gram model of Ukkonen [Ukk92a]. The idea of the maximal matches model is to count the minimal number of occurrences of characters in one string such that if these characters are "crossed out", the remaining subwords are all subwords of the other string. Thus, strings with long common subwords have a small distance. The idea of the q-gram model is to count the number of occurrences of different q-grams in the two strings. Thus, strings with many common q-grams have a small distance. A very interesting aspect is that the maximal matches distance and the q-gram distance of two strings can be computed in time proportional to the sum of the lengths of the two strings, using compact suffix trees.

In the following sections, we consider the three models of string comparison in detail. In section 3.9.4, we review their significance for applications in molecular biology. For the rest of this section let u and v be strings of length m and n, respectively.

3.9.1 The Edit Distance Model

The notion of edit operations was introduced by Ulam [Ula72]. It is the key to the edit distance model.

Definition 3.9.1 An *edit operation* is a pair $(a, b) \in (\mathcal{A}^1 \cup \{\varepsilon\}) \times (\mathcal{A}^1 \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$. □

By abuse of notation, a and b denote *strings* of length ≤ 1. However, if $a \neq \varepsilon$ and $b \neq \varepsilon$, then the edit operation (a, b) is identified with a pair of characters. This motivates the use of the identifiers a and b.

An edit operation (a, b) is usually written as $a \to b$. This reflects the operational view which considers edit operations as rewrite rules transforming a source string into a target string, step by step. In particular, there are three kinds of edit operations:

- $a \to \varepsilon$ denotes the *deletion* of the character a,

- $\varepsilon \to b$ denotes the *insertion* of the character b,
- $a \to b$ denotes the *replacement* of the character a by the character b.

Notice that $\varepsilon \to \varepsilon$ is not an edit operation. Insertions and deletions are sometimes referred to collectively as *indels*. There is no uniform naming for an edit operation of the third type. Some authors use the term replacement [KS83, MM88], as we do. Others instead prefer the terms substitution [Wat89b, Mye91] and change [WF74, Ukk93a].

Sometimes string comparison just means to measure how different strings are. Often it is additionally of interest to analyze the total difference between two strings into a collection of individual elementary differences [KS83]. The most important mode of such analyses is an alignment of the strings.

Definition 3.9.2 An *alignment* A of u and v is a sequence

$$(a_1 \to b_1, \ldots, a_h \to b_h)$$

of edit operations such that $u = a_1 \ldots a_h$ and $v = b_1 \ldots b_h$. □

Obviously, $h \geq \max\{m, n\}$ holds. If A contains at least one indel, we have $h > \max\{m, n\}$. Note that the unique alignment of ε and ε is the empty alignment, that is, the empty sequence of edit operations. An alignment is usually written by placing the characters of the two aligned strings on different lines, with inserted spaces denoting ε. In such a representation, every column represents an edit operation.

Example 3.9.3 The alignment $A = (\varepsilon \to d, b \to b, c \to a, \varepsilon \to d, a \to a, c \to \varepsilon, d \to d)$ of *bcacd* and *dbadad* is written as follows:

$$\begin{array}{cccccccc} & b & c & & a & c & d \\ d & b & a & d & a & & d \end{array}$$

□

Besides alignments, there are also traces and listings as modes of analyses. We do not consider these in the thesis. The reader is referred to [KS83] for a detailed presentation. The notion of optimal alignment requires some scoring or optimization criterion. This is given by a cost function.

Definition 3.9.4 A *cost function* δ assigns to each edit operation $a \to b$, $a \neq b$ a positive real cost $\delta(a \to b)$. The cost $\delta(a \to a)$ of an edit operation $a \to a$ is 0. If $\delta(a \to b) = \delta(b \to a)$ for all edit operations $a \to b$ and $b \to a$, then δ is *symmetric*. If $\delta(a \to b) = 1$, for all edit operations $a \to b$, $a \neq b$ then δ is the *unit cost function*. δ is extended to alignments in a straightforward way: The cost $\delta(A)$ of an alignment $A = (a_1 \to b_1, \ldots, a_h \to b_h)$ is the sum of the costs of the edit operations A consists of. More precisely,

$$\delta(A) = \sum_{i=1}^{h} \delta(a_i \to b_i).$$

□

Definition 3.9.5 The *edit distance* of u and v, denoted by $edist_\delta(u, v)$, is the minimum possible cost of an alignment of u and v. That is,

$$edist_\delta(u, v) = \min\{\delta(A) \mid A \text{ is an alignment of } u \text{ and } v\}.$$

An alignment A of u and v is *optimal* if $\delta(A) = edist_\delta(u, v)$. If δ is the unit cost function, then $edist_\delta(u, v)$ is the *unit edit distance* between u and v. □

If δ is symmetric, then $edist_\delta$ is a metric [WF74]. Note that there can be more than one optimal alignment. The unit edit distance is sometimes called Levenshtein distance.

The edit distance problem is to compute the edit distance and the optimal alignments. A simple algorithm for solving this problem has been independently discovered by many different authors (see [KS83] for an account). In the following, we especially refer to the simplest version which is due to Wagner and Fischer [WF74]. All other differ somewhat. But, as remarked in [KS83], they can be seen as flowing out of the same concept.

The basic idea of the Wagner-Fischer Algorithm is to evaluate the edit distance between longer and longer prefixes of u and v until the final result is obtained. This is stated in the following theorem:

Theorem 3.9.6 Let $E_\delta(i, j) = edist_\delta(u_1 \dots u_i, v_1 \dots v_j)$ for all $i, j, 0 \leq i \leq m$, $0 \leq j \leq n$. Then the following recurrences hold:

$$E_\delta(0, 0) = 0 \tag{3.3}$$

$$E_\delta(i + 1, 0) = E_\delta(i, 0) + \delta(u_{i+1} \to \varepsilon) \tag{3.4}$$

$$E_\delta(0, j + 1) = E_\delta(0, j) + \delta(\varepsilon \to v_{j+1}) \tag{3.5}$$

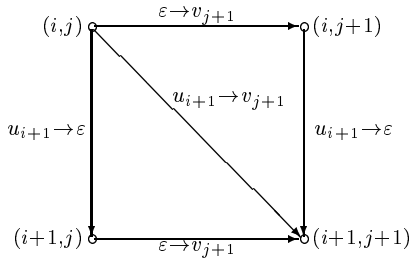
$$E_\delta(i + 1, j + 1) = \min \left\{ \begin{array}{l} E_\delta(i, j + 1) + \delta(u_{i+1} \to \varepsilon) \\ E_\delta(i + 1, j) + \delta(\varepsilon \to v_{j+1}) \\ E_\delta(i, j) + \delta(u_{i+1} \to v_{j+1}) \end{array} \right\} \tag{3.6}$$

Proof See [WF74]. □

The Wagner-Fischer Algorithm computes the $(m + 1) \times (n + 1)$-table E_δ according to the recurrences in Theorem 3.9.6. The computation proceeds row by row, but any other order consistent with the data dependencies of the recurrence is possible as well. $E_\delta(m, n)$ gives the edit distance of u and v. Every entry in E_δ is computed in constant time which leads to an $\mathcal{O}(m \cdot n)$ time complexity. An optimal alignment is recovered by tracing back from the entry $E_\delta(m, n)$ to an entry in its three-way minimum that yielded it, determining which entry gave rise to that entry, and so on back to the entry $E_\delta(0, 0)$. This requires saving the entire table, giving an algorithm that takes $\mathcal{O}(m \cdot n)$ space as well.

Note that if one is interested only in the edit distance of u and v then $\mathcal{O}(n)$ space suffices as one need to keep only the previous row of the table to compute the next row. Using such a distance-only algorithm as a sub-procedure, Hirschberg [Hir75] gave a divide and conquer algorithm that can determine an optimal alignment in $\mathcal{O}(m + n)$ space. This algorithm was given in the context of the longest common subsequence problem, but Myers and Miller [MM88] have shown that it can be applied to most comparison algorithms that have a linear

Figure 3.9: A Part of the Edit Graph $G(u, v)$



space distance-only algorithm. This refinement is very important since space, not time, is often the limiting factor when computing optimal alignments between large strings.

As noted by Myers [Mye91] there is an intuitive graph theoretical formulation of the edit distance problem: The *edit graph* $G(u, v)$ of u and v is an edge labeled graph. The nodes are the pairs (i, j), $0 \leq i \leq m$, $0 \leq j \leq n$. The edges are given as follows:

- For $0 \leq i \leq m-1, 0 \leq j \leq n$ there is a deletion edge $(i, j) \xrightarrow{u_{i+1} \to \varepsilon} (i+1, j)$.
- For $0 \leq i \leq m, 0 \leq j \leq n-1$ there is an insertion edge $(i, j) \xrightarrow{\varepsilon \to v_{j+1}} (i, j+1)$.
- For $0 \leq i \leq m-1, 0 \leq j \leq n-1$ there is a replacement edge $(i, j) \xrightarrow{u_{i+1} \to v_{j+1}} (i+1, j+1)$.

This is illustrated in Figure 3.9.

The central feature of $G(u, v)$ is that each path from $(0, 0)$ to (i, j) is labeled by an alignment of $u_1 \ldots u_i$ and $v_1 \ldots v_j$, and a different path is labeled by a different alignment. An edge $(i', j') \xrightarrow{a \to b} (i, j)$ is *minimizing* if $E_\delta(i, j)$ equals $E_\delta(i', j') + \delta(a \to b)$. A *minimizing path* is any path from $(0, 0)$ to (m, n) that consists of minimizing edges only. In this framework, the edit distance problem means to enumerate the minimizing paths in $G(u, v)$.

Example 3.9.7 Let $u = bcacd$ and $v = dbadad$. Suppose δ is the unit cost function. Then $edist_\delta(u, v) = 4$ and there are the following optimal alignments of u and v.

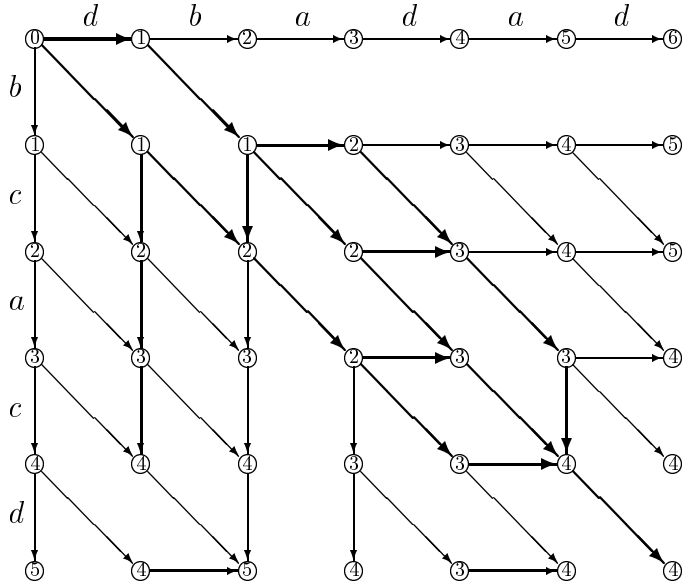
b c a c d d b a d a d	b c a c d d b a d a d	b c a c d d b a d a d	b c a c d d b a d a d	b c a c d d b a d a d	b c a c d d b a d a d	b c a c d d b a d a d

Figure 3.10 shows $G(u, v)$ with all minimizing edges. The minimizing paths are given by the thick edges. Each node is marked by the corresponding edit distance. It is straightforward to read the optimal alignments of u and v from the edit graph. □

In the field of computer science, the edit distance problem mostly occurs in the context of the unit cost function. The longest common subsequence problem [Hir75] and the string-to-string correction problem [WF74] consists of finding alignments with a minimal number of edit operations $a \to b$, $a \neq b$. These problems arose in applications such as comparing the contents of files (see, for instance, the command *diff* in the UNIX[4] operating system)

[4]UNIX is a trademark of Bell Laboratories.

Figure 3.10: The Minimizing Edges and Paths in the Edit Graph $G(bcacd, dbadad)$



and correcting the spelling of words. The restriction to the unit cost function leads to a problem with an interesting combinatorics which was already studied intensively. Let us briefly mention the two main contributions. For ease of presentation we assume $m \leq n$.

Masek and Paterson [MP80] devised an algorithm that computes the unit edit distance of two strings over a finite alphabet in $\mathcal{O}(m \cdot n / \log n)$ time. According to Myers [Mye91], this is the only algorithm that improves on the quadratic worst case time of the Wagner-Fischer Algorithm. Unfortunately, the overhead of Masek and Paterson's algorithm is very high and in practice it leads to a speedup only for very long strings.

Ukkonen [Ukk85a] and Myers [Mye86] have independently devised an algorithm that runs in $\mathcal{O}(n \cdot e)$ time where $e = edist_\delta(u, v)$. The more similar u and v, the more efficient the algorithm becomes. In the expected case, the algorithm takes $\mathcal{O}(n + e^2)$ time, as shown by Myers [Mye86]. However, in the worst case, e is proportional to $m + n$ and so this algorithm does not improve on the quadratic worst case complexity. The reader is referred to Myers [Mye91] and Stephens [Ste94] for a comprehensive survey of techniques for solving the longest common subsequence problem, the string to string correction problem, and related problems.

There are several variations of the Wagner-Fischer Algorithm. Most important is the variation of Sellers [Sel80] which sets the entries in the first row of E_δ to 0. This means that prefix deletions are not charged. Sellers' variation can be used for computing local optimal alignments, and for solving the approximate string searching problem. Other variations of the Wagner-Fischer Algorithm calculate for each table entry some additional information [Ukk93a], or stop computing a column when it is clear that the remaining column values are redundant [Ukk85b]. In section 3.9.5, we develop a unified implementation framework which covers the Wagner-Fischer Algorithm and all these variations.

3.9.2 The Maximal Matches Model

The idea of Ehrenfeucht and Haussler's [EH88] string comparison model is to measure the distance between strings in terms of common subwords. Strings are considered as similar if they have long common subwords. The key to the model is the notion of partition. Recall that u and v are strings of length m and n, respectively.

Definition 3.9.8 A *partition* of v w.r.t. u is a sequence $(w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ of subwords $w_1, \ldots, w_r, w_{r+1}$ of u and characters $c_1, \ldots, c_r$ such that $v = w_1 c_1 \ldots w_r c_r w_{r+1}$. Let $\Psi = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ be a partition of v w.r.t. u. $w_1, \ldots, w_r, w_{r+1}$ are the *submatches* in Ψ. $c_1, \ldots, c_r$ are the *marked characters* in Ψ. The size of Ψ, denoted by $|\Psi|$, is r. $mmdist(v, u)$ is the size of any minimal partition of v w.r.t. u. Following Ukkonen [Ukk92a], we call $mmdist(v, u)$ *maximal matches distance* of v and u. □

The notion of partition subsumes the notion of compatible marking, introduced by Ehrenfeucht and Haussler. More precisely, the marked characters in a partition of v w.r.t. u form a marking of u that makes v compatible with u (cf. [EH88, page 193]). We use partitions instead of compatible markings since it is convenient to also denote the submatches. This convenience will especially be used in section 6.8, when we describe Chang and Lawler's filtering technique for solving the k-differences problem. The function *mmdist* corresponds to the function *diff* in [EH88].

Example 3.9.9 [EH88] Let $v = cbaabdcb$ and $u = abcba$. $\Psi_1 = (cba, a, b, d, cb)$ is a partition of v w.r.t. u, since cba, b, and cb are subwords of u. $\Psi_2 = (cb, a, ab, d, cb)$ is a partition of v w.r.t. u, since cb and ab are subwords of u. It is clear that Ψ_1 and Ψ_2 are of minimal size. Hence, $mmdist(v, u) = 2$. □

There are two canonical partitions.

Definition 3.9.10 Let $\Psi = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ be a partition of v w.r.t. u. If for all $h, 1 \leq h \leq r$, $w_h c_h$ is not a subword of u, then Ψ is the *left-to-right partition* of v w.r.t. u. If for all $h, 1 \leq h \leq r$, $c_h w_{h+1}$ is not a subword of u, then Ψ is the *right-to-left partition* of v w.r.t. u. The left-to-right partition of v w.r.t. u is denoted by $\Psi_{lr}(v, u)$. The right-to-left partition of v w.r.t. u is denoted by $\Psi_{rl}(v, u)$. □

Example 3.9.11 For the strings $v = cbaabdcb$ and $u = abcba$ of Example 3.9.9 we have $\Psi_{lr}(v, u) = \Psi_1$ and $\Psi_{rl}(v, u) = \Psi_2$. □

$\Psi_{lr}(v, u)$ and $\Psi_{rl}(v, u)$ correspond to Ehrenfeucht and Haussler's markings $M_l(v, u)$ and $M_r(v, u)$. As these are of minimal size (see [EH88, Lemma 1.1]), $\Psi_{lr}(v, u)$ and $\Psi_{rl}(v, u)$ are of minimal size, too. Hence, we can conclude $|\Psi_{lr}(v, u)| = mmdist(v, u) = |\Psi_{rl}(v, u)|$. This property leads to a simple algorithm for calculating the maximal matches distance. The partition $\Psi_{lr}(v, u)$ can be computed by scanning the characters of v from left to right, until a prefix wc of v is found such that w is a subword of u, but wc is not. w is the first submatch and c is the first marked character in $\Psi_{lr}(v, u)$. The remaining submatches and marked characters are obtained by repeating the process on the remaining suffix of v, until all of the

characters of v have been scanned. Using the compact suffix tree of u, the longest prefix w of v that is a subword of u, can be computed in $\mathcal{O}(|\mathcal{A}| \cdot |w|)$ time. This gives an algorithm to calculate $mmdist(v,u)$ in $\mathcal{O}(|\mathcal{A}| \cdot (m+n))$ time and $\mathcal{O}(m)$ space.

$\Psi_{rl}(v,u)$ can be computed in a similar way by scanning v from right to left. However, one has to be careful since the reversed scanning direction means to compute the longest prefix of v^{-1} that occurs as subword of u^{-1}. This can, of course, be accomplished by using $cst(u^{-1})$ instead of $cst(u)$.

It is easily verified that $mmdist(u,v) = 1$ and $mmdist(v,u) = 2$ if v and u are as in Example 3.9.9. Hence, $mmdist$ is not a metric on $\mathcal{A}^*$. However, one can obtain a metric as follows:

Theorem 3.9.12 Let $mmm(u,v) = \log_2((mmdist(u,v)+1) \cdot (mmdist(v,u)+1))$. mmm is a metric on $\mathcal{A}^*$.

Proof See [EH88]. □

From the above it is clear that $mmm(u,v)$ can be computed in $\mathcal{O}(|\mathcal{A}| \cdot (m+n))$ steps and $\mathcal{O}(\max\{m,n\})$ space. We will return to the metric mmm when we consider the significance of the different distance models for biological applications.

Next we study the relation of the maximal matches distance and the unit edit distance. We first show an important relation of alignments and partitions.

Lemma 3.9.13 Let A be an alignment of v and u. Then there is an $r, 0 \leq r \leq \delta(A)$, and a partition $(w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ of v w.r.t. u such that w_1 is a prefix and w_{r+1} is a suffix of u.

Proof By structural induction on A. If A is the empty alignment, then $\delta(A) = 0$, $v = u = \varepsilon$, and the statement holds with $r = 0$ and $w_1 = \varepsilon$. If A is not the empty alignment, then A is of the form $(A', b \to a)$ where A' is an alignment of some strings v' and u' and $b \to a$ is an edit operation. Obviously, $v = v'b$ and $u = u'a$. Assume the statement holds for A'. That is, there is an $r', 0 \leq r' \leq \delta(A')$ and a partition $(w_1, c_1, \ldots w_{r'}, c_{r'}, w_{r'+1})$ of v' w.r.t. u' such that w_1 is a prefix and $w_{r'+1}$ is a suffix of u'. First note that w_1 is a prefix of u since it is prefix of u'. There are three cases to consider:

- If $b = \varepsilon$, then $a \neq \varepsilon$ and $\delta(A) = \delta(A') + 1$. Therefore, $v = v'b = w_1c_1 \ldots w_{r'}c_{r'}w_{r'+1}$. If $w_{r'+1}$ is the empty string, then it is a suffix of $u = u'a$. If $w_{r'+1} = wc$ for some string w and some character c, then $v'b = w_1c_1 \ldots w_{r'}c_{r'}wcw'$ where $w' = \varepsilon$ is a suffix of $u = u'a$. Thus, the statement holds with $r = r' + 1 \leq \delta(A)$.

- If $b \neq \varepsilon$ and $a \neq b$, then $\delta(A) = \delta(A') + 1$. Hence, $v = v'b = w_1c_1 \ldots w_{r'}c_{r'}w_{r'+1}bw$ where $w = \varepsilon$ is a suffix of $u = u'a$. Thus, the statement holds with $r = r' + 1 \leq \delta(A)$.

- If $b \neq \varepsilon$ and $a = b$, then $\delta(A) = \delta(A')$. Let $w = w_{r'+1}b$. Then $v = v'b = w_1c_1 \ldots w_{r'}c_{r'}w$, and w is a suffix of $u = u'a$ since $w_{r'+1}$ is a suffix of u'. Thus, the statement holds with $r = r' \leq \delta(A)$. □

The following theorem shows that $mmdist(v,u)$ is a lower bound for the unit edit distance of v and u.

Theorem 3.9.14 Suppose δ is the unit cost function. Then $mmdist(v,u) \leq edist_\delta(v,u)$.

Proof Let A be an optimal alignment of v and u. Then by Lemma 3.9.13 there is a partition Ψ of v w.r.t. u such that $|\Psi| \leq \delta(A)$. Hence, $mmdist(v,u) \leq |\Psi| \leq \delta(A) = edist_\delta(v,u)$. □

A similar result was already obtained in [Ukk92a]. The relation between *mmdist* and $edist_\delta$ suggests to use *mmdist* as a filter in contexts where the unit edit distance is of interest only below some threshold k. In fact, Ukkonen [Ukk92a] as well as Chang and Lawler [CL94] have devised filtering techniques based on maximal matches. They use them to speed up algorithms for the k-differences approximate string searching problem. In section 6.8, we have a closer look on the technique of Chang and Lawler.

3.9.3 The q-Gram Model

Like the maximal matches model, the q-gram model considers common subwords of the strings to be compared. However, while the former model considers subwords of possibly different length, the latter restricts to subwords of a fixed length q. For this section let q be a positive integer. Recall that u and v are strings of length m and n, respectively.

Definition 3.9.15 [Ukk92a] The *q-gram profile* of u is the function $G_q(u) : \mathcal{A}^q \to \mathbb{N}_0$ such that $G_q(u)(w)$ is the number of different positions in u where the string $w \in \mathcal{A}^q$ ends. The *q-gram distance* $qgdist(u,v)$ of u and v is defined by

$$qgdist(u,v) = \sum_{w \in \mathcal{A}^q} |G_q(u)(w) - G_q(v)(w)|. \quad \square$$

One can show that the symmetry and the triangle inequality hold for *qgdist* (cf. [Ukk92a]). The zero property does not hold as shown by the following example. Hence, *qgdist* is not a metric.

Example 3.9.16 Let $q = 2$, $u = aaba$ and $v = abaa$. Then u and v have the same q-gram profile $\{aa \mapsto 1, ab \mapsto 1, ba \mapsto 1, bb \mapsto 0\}$. Hence, the q-gram distance of u and v is 0. □

Ukkonen describes two algorithms for computing the q-gram distance of u and v. We briefly describe them here. The first algorithm encodes each string $w \in \mathcal{A}^q$ into a q-digit integer $\widetilde{w}$ in base $l = |\mathcal{A}|$ (see section 4.3). $\widetilde{w}$ serves as an index into some arrays τ_u and τ_v of size $|\mathcal{A}|^q$. The arrays are used to count the number of occurrences of q-grams of u and v, respectively. In particular, after u and v have been scanned, $\tau_u[\widetilde{w}] = G_q(u)(w)$ and $\tau_v[\widetilde{w}] = G_q(v)(w)$ hold for each $w \in \mathcal{A}^q$. Besides the two arrays, the algorithm constructs a list C consisting of the codes of the q-grams of u and v. This takes $\mathcal{O}(m+n)$ space and time. Initializing each of the array entries with 0 takes $|\mathcal{A}|^q$ time. If each lookup and update operation in τ_u and τ_v can be accomplished in constant time, the profiles are accumulated in the arrays in $\mathcal{O}(m+n)$ time. Merging τ_u and τ_v takes $\mathcal{O}(m+n)$ steps if C is used to skip the zero entries of the arrays. Thus, the overall running time and the space requirement of the algorithm is $\mathcal{O}(m+n+|\mathcal{A}|^q)$. Note that Ukkonen derives a running time of $\mathcal{O}(m+n)$ since he does

not count the time for creating and initializing τ_u and τ_v. Due to the exponential time and space requirement, the applicability of the algorithm is rather limited.

The second algorithm given in [Ukk92a] uses a modified suffix automaton [BBH+85, Cro88]. We outline the algorithm on the basis of compact suffix trees. Let $Gr(u)$ be the set of q-grams of u. First note that $qgdist(u,v)$ can be obtained from the q-gram profiles of u and v restricted to $Gr(u)$, and from the total number of q-grams of v not occurring in u. More precisely:

$$qgdist(u,v) = \sum_{w \in Gr(u)} |G_q(u)(w) - G_q(v)(w)| + \sum_{w \in \mathcal{A}^q \setminus Gr(u)} G_q(v)(w) \tag{3.7}$$

Let $r = |Gr(u)|$. Ukkonen's algorithm computes for each $w \in Gr(u)$ a unique integer code $\widehat{w} \in \{1, \ldots, r\}$, and for each $w \in \mathcal{A}^q \setminus Gr(u)$ the code $\widehat{w} = 0$. $\widehat{w}$ serves as an index into some arrays π_u and π_v of size r and $r+1$, respectively. In a first phase, $T = cst(u)$ is traversed in $\mathcal{O}(|\mathcal{A}| \cdot m)$ time directed by the characters in u. The traverse uses the function $linkloc$ and never reaches a location of depth larger than q. After u has been scanned, the array π_u is computed such that $\pi_u[\widehat{w}] = G_q(u)(w)$ holds for each $w \in Gr(u)$. Moreover, $\widehat{w}$ is stored at location $loc_T(w)$. In a second phase, T is traversed in a similar way, directed by the characters in v. This takes $\mathcal{O}(|\mathcal{A}| \cdot n)$ time and yields the array π_v such that $\pi_v[\widehat{w}] = G_q(v)(w)$ for each $w \in Gr(u)$ and $\pi_v[0] = \sum_{w \in \mathcal{A}^q \setminus Gr(u)} G_q(v)(w)$. Merging π_u and π_v in $\mathcal{O}(r)$ time according to equation (3.7) gives $qgdist(u,v)$. The overall running time of this algorithm is $\mathcal{O}(|\mathcal{A}| \cdot (m+n))$. It requires $\mathcal{O}(m)$ space for T (including the codes), and for the arrays π_u and π_v.

The third algorithm we describe here was developed in cooperation with Robert Giegerich [Gie94b]. It is not published, except in the lecture notes [Gie94a]. The method is similar to the previous algorithm, but does without arrays and can easily be implemented in our functional framework. The idea is to annotate $cst(u\$)$ and $cst(v\$)$ such that the q-gram profile of $u\$$ and $v\$$ can be read from the suffix trees.

Definition 3.9.17 Let T be a compact $\mathcal{A}^+$-tree. For each node $\overline{s}$ in T, $lnum(\overline{s})$ is the number of leaves in the subtree $T_{\overline{s}}$. $lnum$ is an $\mathbb{N}_0$-annotation of T. □

$lnum$ can be computed in $\mathcal{O}(|T|)$ steps by a single traversal of T. Let $T = cst(u\$)$ and $T' = cst(v\$)$ with the annotation $lnum$. It is easily verified that

$$lnum(ceiling(loc_T(w))) = G_q(u\$)(w) \tag{3.8}$$

for each location $loc_T(w)$ of depth q. The corresponding relation holds for v and T'.

Now imagine that T and T' are laid on top of each other. Consider all locations of depth q. There are locations which are common to T and T' and there are locations which either occur in T or in T'. To make this precise, we define three sets $L_\cap$, L_T and $L_{T'}$ as follows:

- $(loc, loc') \in L_\cap \iff (loc, loc') \in locations(T) \times locations(T')$ and there is a $w \in \mathcal{A}^q$ such that $loc = loc_T(w)$ and $loc' = loc_{T'}(w)$.
- $loc \in L_T \iff loc \in locations(T)$, $|loc| = q$ and $loc \notin \{loc \mid (loc, loc') \in L_\cap\}$.
- $loc' \in L_{T'} \iff loc' \in locations(T')$, $|loc'| = q$ and $loc' \notin \{loc' \mid (loc, loc') \in L_\cap\}$.

Each of the elements in $L_\cap$, L_T and $L_{T'}$ contribute to the q-gram distance of $u\$$ and $v\$$, as described in the following algorithm. LOT is an acronym for "lay on top".

Algorithm LOT Enumerate $L_\cap$, L_T and $L_{T'}$ by traversing T and T' in parallel from the *roots* to the locations of depth q (for details see function *pwalk* in section 3.9.5). Compute

$$\begin{aligned} qgdist(u\$, v\$) \;=\; & \sum_{(loc,loc')\in L_\cap} |lnum(ceiling(loc)) - lnum(ceiling(loc'))| + \\ & \sum_{loc\in L_T \cup L_{T'}} lnum(ceiling(loc)) \end{aligned} \tag{3.9}$$

Finally, set $qgdist(u,v) = \begin{cases} qgdist(u\$, v\$), & \text{if } u_{m-q+2}\ldots u_m = v_{n-q+2}\ldots v_n \\ qgdist(u\$, v\$) - 2, & \text{otherwise} \end{cases}$

□

Theorem 3.9.18 Algorithm LOT correctly computes $qgdist(u,v)$ in $\mathcal{O}((|\mathcal{A}|+q)\cdot(m+n))$ time and $\mathcal{O}(m+n)$ space.

Proof From equation (3.8) it immediately follows that $qgdist(u\$, v\$)$ is computed correctly. Notice that

$$qgdist(u\$, v\$) = qgdist(u,v) + \sum_{w\in X} |G_q(u\$)(w) - G_q(v\$)(w)| \tag{3.10}$$

where $X = \{u_{m-q+2}\ldots u_m\$, v_{n-q+2}\ldots v_n\$\}$. If $u_{m-q+2}\ldots u_m = v_{n-q+2}\ldots v_n$, then the second summand in (3.10) is 0. Otherwise, it is 2. This proves the correctness of Algorithm LOT. T and T' including the annotation *lnum* can be computed in $\mathcal{O}(|\mathcal{A}|\cdot(m+n))$ time and $\mathcal{O}(m+n)$ space. No extra space is needed since once an element from $L_\cap$, L_T, or $L_{T'}$ is enumerated its contribution to $qgdist(u\$, v\$)$ can be accumulated. There are $\mathcal{O}(q\cdot m)$ locations of depth $\le q$ in T and $\mathcal{O}(q\cdot n)$ locations of depth $\le q$ in T'. Only these are visited. Hence, the enumeration of the three sets can be accomplished in $\mathcal{O}(q\cdot(m+n))$. $qgdist(u\$, v\$)$ is computed in $\mathcal{O}(m+n)$ according to equation (3.9). Finally, calculating $qgdist(u,v)$ from $qgdist(u\$, v\$)$ takes $\mathcal{O}(q)$ steps. □

Note that we can vary Algorithm LOT such that $qgdist$ is directly computed form $cst(u)$ and $cst(v)$. This variation requires to annotate each node $\overline{s}$ in $cst(u)$ by the number of positions in u where s ends. $cst(v)$ is annotated correspondingly. This annotation can not efficiently be constructed after the suffix trees have been computed. This is because if s is a prefix of a nested suffix sx of u, then the corresponding occurrence of s is not witnessed by a leaf $\overline{sx}$ in $cst(u)$. However, the annotation is available in the lazy suffix tree algorithm. Consider the function *lazytree* defined on page 52. Suppose the local function *subtree* constructs a node $\overline{s}$ in $cst(u)$ from the list ss. Then the number of positions in u where s occurs is equal to $|ss|$. By a simple modification of the lazy suffix tree algorithm, the length of ss can be made available to the function *subtree*. This allows to compute the annotation at virtually no cost. We do not know how to modify Ukkonen's online algorithm such that it simultaneously computes the annotation.

The variation of Algorithm LOT is very interesting. Like the second algorithm of Ukkonen, it requires to construct only the nodes of depth $\le q$. This may improve the practical running time considerably.

Like the maximal matches distance, the q-gram distance provides a lower bound for the unit edit distance.

Theorem 3.9.19 Let δ be the unit cost function. Then $qgdist(u,v)/(2 \cdot q) \leq edist_\delta(u,v)$.

Proof See [JU91, Ukk92a].

Ukkonen exploits this relation to speed up approximate string searching algorithms which are based on the unit edit distance. In particular, he uses the q-gram distance as a filter for determining subwords of the input string where an approximate match may occur. For details see [JU91, Ukk92a].

3.9.4 Significance of the Models in Molecular Biology

In molecular biology, strings occur as a linear representation of information recorded in a genetic macromolecule, such as DNA, RNA, or proteins. The main use of string comparison in that context is to detect homologies between such biosequences. Two biosequences have a high degree of homology if the differences that have developed between them are relatively minor. Knowledge of homology sheds light on evolution and on the structure and function of organisms. In particular, today it is standard practice to compare new biosequences against the current databases to look for homologies that will hopefully give significant clues to their function [AGM+89].

It is especially the edit distance model that provides the basis for most programs comparing biosequences. According to [KS83], the enormous success is due to the following facts:

- Deletions, insertions and replacements are accepted as a fairly accurate model of the evolutionary process at the nucleotide level. Hence, the result of a comparison often conforms closely to what biologists are likely to consider appropriate and realistic.
- The cost function provides a simple and well-motivated criterion for choosing among homologies.
- The Wagner-Fischer Algorithm for solving the edit distance problem is simple and practical. The most important point, however, is that the algorithm always finds the best homology out of all possible homologies between the biosequences.
- The edit distance model and the Wagner-Fischer Algorithm extends in a natural and consistent way to a wide range of problems occurring in the comparison of biosequences. In particular, these are the problem of handling gap costs [NW70], the problem of simultaneously comparing several biosequences [CL88, AL89], and the problem of approximate searching in sequence databases [Mye94b].

Note that in the field of molecular biology the Wagner-Fischer Algorithm is often referred to as *the* dynamic programming algorithm. We have tried to avoid this confusion. Following Myers [Mye91], we emphasize that dynamic programming is a general computational paradigm of wide applicability. A problem is solved by dynamic programming if the answer can be efficiently determined by computing a table of optimal answers to progressively larger

and larger subproblems. The principle of optimality requires that the optimal answer to a given problem can be expressed in terms of optimal answers to smaller subproblems. The edit distance follows this principle which was formally stated in Theorem 3.9.6.

The most important disadvantage of the edit distance is that it can only be computed in more or less quadratic time. The maximal matches and the q-gram distance can be computed faster, however, for the cost of less flexibility. Except for the choice of different values of q, there is no parameter, comparable to the cost function in the edit distance model, that can be used to "calibrate" the distances depending on the specific biological application.

As noted earlier, there are situations where the edit distance model is inappropriate. Consider some long strings x, y and z and a single character a. The edit distance between xyz and $xayz$ is usually very small, but between xyz and zxy it is usually very large. A good example for such a situation occurs in molecular biology: For instance, xyz and zxy may be two different linear representations of the same piece of circular DNA, obtained by cutting the strand in two different places [CHM+86]. The metric mmm and the q-gram distance (if q is chosen appropriately) are relatively invariant from the positions where x, y, and z occur. Hence, both distance models can cope with this situation.

Unfortunately, for both the maximal matches and the q-gram comparison model there is no convenient mode to analyze the total difference into a collection of individual differences. Without such a representation it is doubtful that the two distance models will find a wide range of applicability in string comparisons for molecular biology.

Recently, Berndt [Ber95] studied the different string comparison models by analyzing the phylogeny of bacteria. The main results of the study are as follows: The edit distance and the q-gram distance (for $q = 5, 7, 9$) allows to reconstruct the expected evolutionary relationships, while the metric mmm does not.

3.9.5 Implementation

Edit Distance

The main goal of this section is to develop a unified framework for implementing variations of the Wagner-Fischer Algorithm. Using a polymorphic type for representing the dynamic programming table, we can cope with the different kinds of entries computed by the variations. Using higher order functions, we can abstract from the different ways new table entries are computed from previous table entries.

Edit operations are implemented by a type *editoperation* which introduces three constructors D, I, and R for a deletion, insertion and replacement operation. Alignments are lists of edit operations.

```
editoperation α ::= D α | I α | R α α
alignment α == [editoperation α]
```

Note that both types are polymorphic. We can use them to represent edit operations and alignments over some arbitrary alphabet.

Example 3.9.20 The alignment $A = (\varepsilon \to d, b \to b, c \to a, \varepsilon \to d, a \to a, c \to \varepsilon, d \to d)$ of *bcacd* and *dbadad* (see Example 3.9.3) is represented by the following list:

```
[I d, R b b, R c a, I d, R a a, D c, R d d]
```

□

Since the constructor expressions are hard to read, we introduce a function *showalign* that converts an alignment into the usual representation in lines. *showalign* proceeds from right to left. It yields two lines *line* and *line'* which are concatenated with a newline character in between. Each character in *line* is aligned with a character in *line'*. An edit operation $(D\ a)$ means that the character a in *line* is aligned with a space in *line'*. $(I\ b)$ means that a space in *line* is aligned with the character b in *line'*. $(R\ a\ b)$ means that the character a in *line* is aligned with the character b in *line'*.

```
showalign::alignment char→[char]
showalign alignment = line ++ "\n" ++ line'
                      where (line,line') = foldr front ([],[]) alignment
                            front (D a) (line,line') = (a:line,' ':line')
                            front (I b) (line,line') = (' ':line,b:line')
                            front (R a b) (line,line') = (a:line,b:line')
```

For the sake of good readability we introduce a type synonym for cost functions.

```
costfunction α == (editoperation α)→num
```

The unit cost function is given as follows:

```
unitcost::costfunction α
unitcost (R a a) = 0
unitcost eop = 1
```

A straightforward implementation of the Wagner-Fischer Algorithm represents the table E_δ by a two-dimensional array. The recurrences of Theorem 3.9.6 can then be translated directly into some iterations and assignment operations. We do not adopt this solution. Instead, we represent the table by a list of columns, and the columns by a list of values of some arbitrary type β.

```
table β == [column β]
column β == [β]
```

The table entries are usually of type *num*, but we also consider variations with pairs and lists of pairs.

The variations of the Wagner-Fischer Algorithm we implemented, compute a table column by column in a unified way. In particular, every variation is specified by a *table specification*, that is, a pair $(\mathit{firstcol}, \mathit{nextcol})$ of type

```
tablespec α β == (column β,column β→α→column β)
```

firstcol is the first column of the table. *nextcol* is a function that takes the previous column of the table and the actual character, and produces the next column of the table. As we will see later, an expression for computing the first column is easily obtained. The complicated part of a table specification is the function *nextcol*.

Recall that u and v are strings of length m and n, respectively. The nucleus of our implementation is a higher order function *absnextcol*, which abstracts from the differences of the variations. In particular, *absnextcol* takes three functions *firstentry*, *join*3, and *join*2 as arguments:

1. *firstentry* computes the first entry of a column.

2. *join*3 computes a new entry from three previous entries.

3. *join*2 computes a new entry from two previous entries plus the rest of the column. Note that this function is only used for the cutoff variations of the Wagner-Fischer Algorithm described in sections 6.1, 6.4, and 6.2.5. These variations never evaluate the bottom portion of a column if the corresponding entries can be inferred to be redundant.

absnextcol has some more arguments: The string u, the previous column *col* of the table, and a character $b = v_j$ for some $j, 1 \leq j \leq n$.

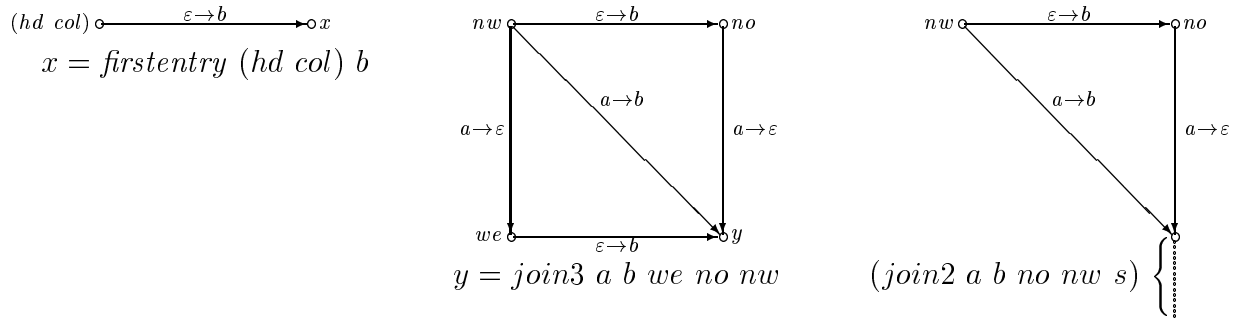
```
absnextcol::(β→α→β)→(α→α→β→β→β→β)→(α→α→β→β→(string α)→[β])→
            (string α)→(column β)→α→(column β)
absnextcol firstentry join3 join2 u col b
  = x:nextval u col x
    where x = firstentry (hd col) b
          nextval [] restcol no = []
          nextval (a:s) (nw:we:restcol) no = y:nextval s (we:restcol) y
                                             where y = join3 a b we no nw
          nextval (a:s) [nw] no = join2 a b no nw s
```

absnextcol computes at most $m + 1$ entries of a column as follows:

1. The first entry x of a column is obtained from the first entry (*hd col*) of the previous column, and the character b. This is illustrated in Figure 3.11 by an insertion edge occurring in the first row of the edit graph $G(u, v)$ (see section 3.9.1).

2. The rest of the column is computed by a recursively defined function *nextval*. The first argument of *nextval* is a suffix of u, the second is a suffix of the previous column, and the third is the previously computed value *no* of the actual column. If the suffix of u is empty, then there is no value to be computed. Now consider the situation where the suffix of u is of the form $(a : s)$ and there are at least two remaining values nw and we of the previous column. Then a new entry y is computed by applying *join*3 to the characters a, b, and the three table entries to the "west", the "north" and the "northwest". This is illustrated by a part of the edit graph $G(u, v)$ as shown in Figure 3.11.

Figure 3.11: Graphical Explanation of *absnextcol*



3. Suppose the suffix of u is of the form $(a : s)$ and there is only one remaining value nw of the previous column. (Recall that this case occurs only for the cutoff variations.) Then $join2$ is applied to the characters a and b, to the table entries in the "north" and the "northwest" and the remaining suffix s of u. This is illustrated in Figure 3.11 by a part of the edit graph $G(u, v)$. Note that $join2$ returns a list of values, that is, the remaining entries of the column to be computed. This list can be empty.

absnextcol does without any index operations. The computation is organized in such a way that those entries of a column necessary for the computation of the next entry, are always at hand. They are either selected by a *hd*-operation (case 1) or they occur as arguments of the function *nextval* (cases 2 and 3). Thus, *absnextcol* computes each column entry in constant time, and therefore the choice to take lists for the main data structure does not have a negative effect on the asymptotic efficiency of the implementation.

The geographical naming convention for the table entries is adopted from Allison [All92]. The structure of *absnextcol* is very similar to Allison's functional program for computing the unit edit distance (see [All92, Algorithm 2]). However, our approach is more general. *absnextcol* implements the central idea of all variations of the Wagner-Fischer Algorithm considered in the thesis. By instantiating the first three arguments of *absnextcol*, we obtain a *nextcol*-function useful for solving a specific string comparison or approximate string searching problem. This is shown by the following table specification for the Wagner-Fischer Algorithm.

```
edisttable::(costfunction α)→(string α)→tablespec α num
edisttable delta u
  = (firstcol,nextcol)
    where firstcol = 0:[no + delta (D a) | (no,a)←zip2 firstcol u]
          firstentry we b = we + delta (I b)
          join3 a b we no nw
            = min2 (min2 (we+delta (I b)) (no+delta (D a))) (nw+delta (R a b))
          nextcol = absnextcol firstentry join3 undef u
```

For each cost function *delta* and each string u, the expression (*edisttable delta u*) returns a table specification for the Wagner-Fischer Algorithm. The first column of the table is computed by the expression *firstcol*. The first entry in *firstcol* is 0, according to equation

(3.3) of Theorem 3.9.6. The other entries of *firstcol* are obtained by adding the cost of a deletion operation to the previously computed entry to the "north" (see equation (3.4) of Theorem 3.9.6). Note the use of programming with unknowns: *firstcol* is defined and used simultaneously. Since the first value 0 of *firstcol* is given, this leads to a welldefined result.

By instantiating the first three arguments of *absnextcol*, we derive a function *nextcol* which implements equations (3.5) and (3.6) of Theorem 3.9.6. The first entry of each column is obtained by adding the cost of an insertion operation to the entry in the "west" (see *firstentry*). Three entries are joined by adding the cost of the corresponding edit operation to each and taking the minimum value (see $join3$). Note that the columns computed are of the same length. Hence, the situation where exactly two values must be joined never occurs. In other words, $join2$ is never called and we can set the third argument of *absnextcol* to *undef*.

Using the table specification *edisttable*, we obtain a function *edist* that computes the edit distance of two strings u and v with respect to a cost function *delta* in $\mathcal{O}(m \cdot n)$ time. Starting with the first column, all other columns are computed by applying *nextcol*. The edit distance is the last entry of the last column computed. Note that we do not use explicit recursion in *edist*, but the built-in function *foldl*. According to [Tur89], *foldl* forces *nextcol* to evaluate its first argument which is the column in our case. Hence, *edist* needs only $\mathcal{O}(m)$ space.

```
edist::(costfunction α)→(string α)→(string α)→num
edist delta u v = last (foldl nextcol firstcol v)
                  where (firstcol,nextcol) = edisttable delta u
```

If we assume the unit cost model, then *edisttable* can be considerably simplified. Since the cost for deleting and inserting a character is 1, the list $[0...m]$ is the first column of the table. Moreover, the first entry of each column can be obtained from the first entry of the previous column by adding 1. To speed up the function $join3$, one exploits that $nw \leq we + 1$ and $nw \leq no + 1$ hold (see Lemma 6.3.3).

1. If $a = b$, then $\min\{nw, no+1, we+1\} = nw$. That is, we and no need not be evaluated.
2. If $a \neq b$ and $no < nw$, then $no \leq we$ and therefore $\min\{nw+1, no+1, we+1\} = no+1$. That is, we need not be evaluated.
3. If $a \neq b$ and $no \geq nw$, then $\min\{nw + 1, no + 1, we + 1\} = \min\{we, nw\} + 1$.

This suggests the following function $ujoin3$:

```
ujoin3::α→α→num→num→num→num
ujoin3 a b we no nw = nw,              if a = b
                    = 1 + no,          if no < nw
                    = 1 + min2 we nw, otherwise
```

The restriction to the unit cost function leads to the following table specification.

```
uedisttable::(string α)→tablespec α num
uedisttable u = (firstcol,nextcol)
                where firstcol = [0..#u]
                      firstentry we b = 1 + we
                      nextcol = absnextcol firstentry ujoin3 undef u
```

The unit edit distance of u and v is computed by the function *uedist* in $\mathcal{O}(m \cdot n)$ time. *uedist* is equivalent to Allison's [All92] functional program for computing the unit edit distance.

```
uedist::(string α)→(string α)→num
uedist u v = last (foldl nextcol firstcol u)
             where (firstcol,nextcol) = uedisttable v
```

Note that Allison also gives a functional program that implements the algorithm of [Ukk85a] and Myers [Mye86] mentioned above. The program computes the unit edit distance of u and v in $\mathcal{O}(\min\{m,n\} \cdot edist_\delta(u,v))$ time by exploiting the laziness. It efficiently implements a greedy strategy that requires added complication when the algorithm is written in an imperative language. Unfortunately, Allison's program does not fit into our framework since it computes the table entries diagonal by diagonal. Therefore, we will not consider it here.

We now study the problem of computing the optimal alignments. If the table E_δ of the Wagner-Fischer Algorithm is represented by a two dimensional array, it can be checked in constant time, whether or not an edge $(i',j') \xrightarrow{a \to b} (i,j)$ in the edit graph $G(u,v)$ is minimizing. This gives a simple back-tracking algorithm that computes optimal alignments by enumerating the minimizing paths in the graph $G(u,v)$. The running time of this algorithm is proportional to the sum of the length of the optimal alignments.

Adopting this algorithm to our framework, would lead to an inefficient program, due to the list representation of the main data structure. Our approach for computing optimal alignments is more general. We apply the "list of successes" technique of Wadler [Wad85]. In each entry (i,j) of the table, we calculate the list of all alignments of $u_1 \ldots u_i$ and $v_1 \ldots v_j$ ordered by their costs. The optimal alignments can then easily be obtained by taking some prefix of this list. The following table specification clarifies our approach.

```
aligntable::(costfunction α)→(string α)→tablespec α [(num,alignment α)]
aligntable delta u
  = (firstcol,nextcol)
    where add e eop = [(cost + delta eop,eop:alignment) | (cost,alignment)←e]
          firstcol = [(0,[])]:[add no (D a) | (no,a)←zip2 firstcol u]
          firstentry we b = add we (I b)
          join3 a b we no nw
            = merge (merge (add we (I b)) (add no (D a))) (add nw (R a b))
          nextcol = absnextcol firstentry join3 undef u
```

The table entries are lists of pairs. Each pair consists of an alignment with its cost in the first component. The functions *firstcol*, *firstentry*, and *join*3 are defined similarly as in *edisttable*. The differences are as follows:

- The first entry of *firstcol* is the list consisting of the empty alignment with cost 0.

- The function + is replaced by the function *add* which can be considered as the extension of + on values of type $[(num, alignment\ \alpha)]$. *add* takes all pairs of the form $(cost, alignment)$ from some entry e and adds an edit operation *eop* to such a pair. In this way, the alignments are constructed in reverse order.
- The function $min2$ is replaced by the function *merge* which produces a single ordered list from its ordered arguments (see section 2.1.2).

Using the table specification *aligntable*, we obtain a function *enumalignments* that enumerates the alignments of u and v w.r.t. a cost function *delta*.

```
enumalignments::(costfunction α)→(string α)→(string α)→[(num,alignment α)]
enumalignments delta u v = last (foldl nextcol firstcol v)
                           where (firstcol,nextcol) = aligntable delta u
```

In order to select appropriate alignments from the list of all alignments, we use the function *selectalignments*. The first argument of *selectalignments* is a function *select* which specifies the alignments to be selected from the second argument *alignments*. The edit distance *edist* is the first component of the first element of *alignments*. Using *takewhile*, we obtain a maximal prefix of *alignments* such that for each element $(cost, alignment)$ of this prefix, the expression $(select\ edist\ cost)$ evaluates to true. Finally, from each selected pair the second component is taken and reversed.

```
selectalignments::(num→num→bool)→[(num,alignment α)]→[alignment α]
selectalignments select alignments
  = map (reverse.snd) (takewhile (select edist.fst) alignments)
    where edist = fst (hd alignments)
```

To compute optimal alignments we only have to provide a function that selects all alignments whose costs are equal to the edit distance:

```
optimal::(costfunction α)→(string α)→(string α)→[alignment α]
optimal delta u v = selectalignments (=) (enumalignments delta u v)
```

The running time of *optimal* is $\mathcal{O}(m \cdot n + mergetime)$, where *mergetime* is the time used by the function *merge* (see *join*3 above). First note that *merge* applies a predefined polymorphic comparison function $\leq$ to compare some pairs $(cost, alignment)$ and $(cost', alignment')$ (cf. the definition of *merge* in section 2.1.2). If $(cost \neq cost')$, then the comparison immediately yields true or false. If $(cost = cost')$, then additionally *alignment* and *alignment*$'$ are compared. Since the first edit operation in *alignment* is different from the first edit operation in *alignment*$'$, the comparison immediately yields true or false. In other words, the comparison operation performed by *merge* always takes constant time. This implies that *mergetime* is proportional to the number of calls of the function *merge*. The latter is bound by the sum of the lengths of all optimal alignments. Thus, *optimal* achieves the same asymptotic running time as the back-tracking algorithm described above.

Our approach requires lazy evaluation. For each table entry (i, j) only a prefix of all alignments of $u_1 \ldots u_i$ and $v_1 \ldots v_j$ is evaluated, namely the optimal alignments. Thus, the space

requirement is proportional to the sum of the length of the optimal alignments. In a language with eager evaluation, our approach would clearly lead to exponential space and time requirements.

In biological applications, it is sometimes necessary to compute suboptimal alignments, that is, alignments whose costs lie in the ϵ-neighborhood of the edit distance, for some $\epsilon > 0$ (see [NB93]). Reusing the functions *enumalignments* and *selectalignments* we can easily solve this problem by an appropriate *select*-function.

```
suboptimal::num→(costfunction α)→(string α)→(string α)→[alignment α]
suboptimal epsilon delta u v = selectalignments select (enumalignments delta u v)
                               where select edist cost = epsilon >= cost - edist
```

Note that our approach of computing suboptimal alignments allows a maximum of flexibility since the enumeration of the alignments and the selection of suboptimal alignments are logically separated. The laziness guarantees the time and space efficiency of the approach. In this sense, lazy evaluation provides a very powerful mechanism to "glue" individual program parts together, as remarked by Hughes [Hug89]. In an eager language, the separation of the different tasks is usually not achieved. The program code for enumerating the alignments becomes intertwined with the code for selecting suboptimal alignments, resulting in a monolithic program.

Maximal Matches Distance

Suppose $\Psi_{lr}(v,u) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ is the left-to-right partition of v w.r.t. u (see Definition 3.9.8). Moreover, let $T = cst(u)$ and $loceps = loc_T(\varepsilon)$. $(lrpartition\ loceps\ v)$ computes in $\mathcal{O}(|\mathcal{A}| \cdot n)$ time and $\mathcal{O}(m)$ space the list

$$[(vnodes_1, loc_1), \ldots, (vnodes_r, loc_r), (vnodes_{r+1}, loc_{r+1})] \tag{3.11}$$

where $loc_h = loc_T(w_h)$, and $vnodes_h$ is the list of nodes visited while traversing T from the *root* to the location loc_h. *lrpartition* uses a function *locmax* which calls *scanprefix'* to determine the values $vnodes_h$, loc_h, and the remaining suffix $cx = c_h w_{h+1} \ldots w_r c_r w_{r+1}$ of v.

```
lrpartition::(location α β)→(string α)→[([ctree α β],location α β)]
lrpartition loceps v
  = locmax (scanprefix' loceps v)
    where locmax (vnodes,loc,c:x) = (vnodes,loc):locmax (scanprefix' loceps x)
          locmax (vnodes,loc,[]) = [(vnodes,loc)]
```

The list (3.11) contains valuable information which is used in other contexts (see section 6.8). To determine $mmdist(v,u)$, we only need the length of this list. Moreover, it is not necessary to annotate the compact suffix tree. Therefore, we let the second argument (the annotation function) of *cst* undefined.

```
mmdist::(string α)→(string α)→num
mmdist v u = #(lrpartition (LocN (cst (u,#u) undef)) v) - 1
```

Note that due to the laziness the elements of the list (3.11) are not evaluated for computing $mmdist(v,u)$. With a function ($logbase$ 2), which calculates the logarithm to base 2, an implementation of the function mmm is now straightforward. Note that $logbase$ uses the predefined function log which returns the natural logarithm.

```
mmm::(string α)→(string α)→num
mmm u v = logbase 2 ((mmdist u v + 1) * (mmdist v u + 1))
          where logbase base i = log i/log base
```

mmm takes $\mathcal{O}(|\mathcal{A}| \cdot (m+n))$ time and $\mathcal{O}(\max\{m+n\})$ space. Hence, our implementation is optimal.

q-Gram Distance

In this section, we give an implementation of Algorithm LOT. We do not implement the variation described in section 3.9.3 since it only works in conjunction with a modified lazy suffix tree algorithm which computes for each node $\overline{s}$ in $cst(x)$ the number of occurrences of s in x.

Consider a compact $\mathcal{A}^+$-tree T and suppose $\overline{v}$ is a node in T such that es represents the set of edges outgoing from $\overline{v}$. The function $getlnum$ computes $lnum(\overline{v})$ (see Definition 3.9.17). If $\overline{v}$ is a leaf, then $es = [\,]$ and $lnum(\overline{v}) = 1$. Otherwise, $lnum(\overline{v})$ is the sum of the $lnum$-annotations of the nodes directly below $\overline{v}$. $getlnum$ takes $\mathcal{O}(|T|)$ time to construct the annotation $lnum$ since each edge of T is inspected exactly once.

```
getlnum::annotationfunction α num
getlnum d es = 1,                                        if es = []
             = sum [lnum | (w,N es' link lnum)←es], otherwise
```

Let loc be a location in T. The function $scanone$ enumerates all locations which can be reached from loc by scanning one character of an edge label in T. More precisely, if $loc = loc_T(w)$ for some $w \in words(T)$, then the expression ($scanone\ loc$) returns the ordered list $[(a, loc_T(wa)) \mid \exists a \in \mathcal{A} : wa \in words(T)]$ in time proportional to its length.

```
scanone::(location α β)→[(α,location α β)]
scanone (LocN node)
  = map down (seledges node)
    where down ((w,k),node')
            = (hd w,LocE node (w,1) (drop 1 w,k-1) node'), if isleaf node' ∨ k > 1
            = (hd w,LocN node'),                           otherwise
scanone (LocE node (v,j) (w,i) node')
  = [],                                                  if i = 0
  = [(hd w,LocE node (v,j+1) (drop 1 w,i-1) node')], if isleaf node' ∨ i > 1
  = [(hd w,LocN node')],                                 otherwise
```

The function $scanjust$ iterates $scanone$. More precisely, if $loc = loc_T(w)$ for some $w \in words(T)$ then ($scanjust\ k\ loc$) returns the list $[loc_T(wx) \mid \exists x \in \mathcal{A}^k : wx \in words(T)]$ in time proportional to the number of visited locations.

```
scanjust::num→(location α β)→[location α β]
scanjust (i+1) loc = [loc'' | (a,loc')←scanone loc; loc''←scanjust i loc']
scanjust 0 loc = [loc]
```

Now suppose $T = cst(u\$)$ and $T' = cst(v\$)$. To represent the disjoint union of the sets $L_\cap$, L_T, and $L_{T'}$ (see section 3.9.3), we introduce the type *oneortwo*:

```
oneortwo γ ::= One γ | Two γ γ
```

Each pair $(loc, loc') \in L_\cap$ is represented by the expression $(Two\ loc\ loc')$. Each location $loc \in L_T \cup L_{T'}$ is represented by the expression $(One\ loc)$. Hence, the union of $L_\cap$, L_T and $L_{T'}$ is represented by a list *repL* of type $[oneortwo\ (location\ \alpha\ num)]$.

The function *pwalk* implements a parallel walk of T and T' which yields the list *repL* in $\mathcal{O}(q \cdot (m+n))$ time.

```
pwalk::num→(location α β)→(location α β)→[oneortwo (location α β)]
pwalk (i+1) loc loc' = mergelocs i (scanone loc) (scanone loc')
pwalk 0 loc loc' = [Two loc loc']
```

pwalk is called with three arguments k, *loc*, and *loc'* such that $0 \leq k \leq q$, $loc = loc_T(w)$, and $loc' = loc_{T'}(w)$ for some $w \in \mathcal{A}^{q-k}$. It enumerates all pairs $(loc_T(wx), loc_{T'}(wx)) \in L_\cap$ and all locations $loc_T(wx) \in L_T$ and $loc_{T'}(wx) \in L_{T'}$ for some $x \in \mathcal{A}^k$. If $k = 0$, then $w \in \mathcal{A}^q$ and $(loc, loc') \in L_\cap$ is the only element enumerated. If $k = i + 1$ for some $i \geq 0$, then $(scanone\ loc)$ and $(scanone\ loc')$ is evaluated. This yields two ordered lists which are merged by a function *mergelocs*.

```
mergelocs::num→[(α,location α β)]→[(α,location α β)]→[oneortwo (location α β)]
mergelocs i ((a,loca):locs) ((b,locb):locs')
  = pwalk i loca locb ++ mergelocs i locs locs',                    if a = b
  = map One (scanjust i loca) ++ mergelocs i locs ((b,locb):locs'), if a < b
  = map One (scanjust i locb) ++ mergelocs i ((a,loca):locs) locs', otherwise
mergelocs i locs [] = [One loc | (a,loca)←locs; loc←scanjust i loca]
mergelocs i [] locs' = [One loc | (b,locb)←locs'; loc←scanjust i locb]
```

Suppose the second and the third argument of *mergelocs* are of the form $((a, loca) : locs)$ and $((b, locb) : locs')$, respectively. If $a = b$, then $loca = loc_T(wa)$ and $locb = loc_{T'}(wa)$. Hence, *pwalk* is called recursively with the three arguments i, *loca*, and *locb* and *mergelocs* is applied to the remaining lists *locs* and *locs'*. If $a < b$, then T' does not contain a location for wa. Hence, $(scanjust\ i\ loca)$ enumerates all elements $loc_T(wax') \in L_T$ for some $x' \in \mathcal{A}^i$. Correspondingly, for the case $a > b$. The cases where either the second or the third argument of *mergelocs* is empty, are analogous.

The use of the function *scanone* considerably simplifies the implementation of Algorithm LOT. In a previous implementation without this function, we had a dozen of case distinctions in *mergelocs* and *pwalk*. Case distinctions concerning edge locations and node locations are now gathered in *scanone*.

The function *loc2num* extracts from each element in *repL* its contribution to the q-gram distance of $u\$$ and $v\$$ (see equation 3.9 in Algorithm LOT). Note that *abs* is the predefined function for computing the absolute value of a number.

```
loc2num::(oneortwo (location α num))→num
loc2num (One loc) = seltag (ceiling loc)
loc2num (Two loc loc') = abs (seltag (ceiling loc) - seltag (ceiling loc'))
```

The function *qgdist* computes the q-gram distance of u and v according to algorithm LOT.

```
qgdist::α→num→(string α)→(string α)→num
qgdist sentinel q u v
  = sum (map loc2num repL),     if drop (m-q+1) u = drop (n-q+1) v
  = sum (map loc2num repL) - 2, otherwise
    where m = #u
          n = #v
          loceps = LocN (cst (u++[sentinel],m+1) getlnum)
          loceps' = LocN (cst (v++[sentinel],n+1) getlnum)
          repL = pwalk q loceps loceps'
```

Initially, *qgdist* constructs T and T' including the annotation *lnum*. Then it computes $loceps = loc_T(\varepsilon)$ and $loceps' = loc_{T'}(\varepsilon)$ and evaluates the expression (*pwalk q loceps loceps'*) yielding the list *repL*. Finally, $qgdist(u\$, v\$)$ is computed according to equation (3.9) in Algorithm LOT. To obtain the string $u_{m-q+2} \ldots u_m$, the first $m - q + 1$ elements of u are dropped. Correspondingly, $v_{n-q+2} \ldots v_n$ is obtained. From the above it is clear that the overall running time of the function *qgdist* is $\mathcal{O}((|\mathcal{A}| + q) \cdot (m + n))$. Due to the laziness, only one element of the list *repL* is stored at any time of the computation. Hence, the space requirement of *qgdist* is $\mathcal{O}(m + n)$ and our implementation is optimal.

3.10 Summary

In this chapter, we have established a powerful string processing machinery. We have shown how to represent and construct various forms of $\mathcal{A}^+$-trees and dynamic programming tables in a unified way. Using polymorphic types, we were able to abstract from the different underlying alphabets, the different degrees of compression, and the different forms of table entries. Using higher order functions, we were able to abstract from the different forms of application-specific annotations, and the diverse ways a new table entry is obtained from previous table entries. Our string processing machinery consists of the following main components:

- A data type *location* with the functions *getloc*, *scanprefix*, and *linkloc*. This data type allows a convenient and concise implementation of algorithms that construct or traverse suffix trees.
- A higher order function *cst* to compute suffix trees including suffix links and annotations in a unified way.
- A higher order function *absnextcol* which provides a flexible framework to implement various forms of the Wagner-Fischer Algorithm in a unified way.

In the following chapters, we will apply these components in many different contexts.

Chapter 4

Exact String Searching

The exact string searching problem occurs in many different contexts, like text editing, data retrieval, and symbol manipulation. It is one of the best studied problems in computer science. We present some well-known and one widely unknown exact string searching algorithm in detail, and show how to implement the algorithms in a unified functional framework. We begin with a precise definition of the problem.

Definition 4.0.1 Suppose the following items are given:

- A string $p \in \mathcal{A}^*$ of length m.
- A string $t \in \mathcal{A}^*$ of length n.

$\mathcal{A}$ is the *input alphabet*, p is the *pattern*, and is t the *input string*. The *exact string searching problem* is to enumerate all positions in t where p ends. These positions are referred to as *solutions* to the exact string searching problem. □

Let $\mathcal{A}_p$ be the set of characters in p. In general, m is much smaller than n which means that $\mathcal{A}_p$ contains only a small fraction of the input characters. To state efficiency results, we sometimes have to refer to this fraction $\mathcal{A}_p$ explicitly.

We assume the input string to be given online. Each considered exact string searching algorithm scans t from left to right. At each time the access to t is restricted to a small sliding window of size $\mathcal{O}(m)$. Our functional implementations preserve these properties of the algorithms. In particular, t is represented by a lazy list, that is, a stream of input characters. The length of this list is never computed. However, when we give efficiency results for our programs, we always refer to an input string of length n. An important virtue of our functional implementations is that, except for the three algorithms of Boyer-Moore type, an explicit buffering mechanism is not required to achieve the space efficiency of $\mathcal{O}(m)$. Lazy evaluation and the memory management of the Miranda system guarantee that only the needed portion of the input string is stored at any time.

The problem formulation of Definition 4.0.1 is consistent with [BY89c, CLR90, CL90, Sun90, HD91]. Some authors restrict themselves to the simplified form of the exact string searching problem such that only the first occurrence of p in t has to be found [Hor80, Sed88, Aho90, Pir92, Ste94]. They argue that a program for the simplified form can easily be extended

to solve the general problem. We emphasize that the lazy functional programs presented in this chapter can be used to solve both forms of the problem, without changing their code. This is due to the compositionality: For each of our functions of the form

```
f::(string α)→(string α)→[num]
f p t = ...
```

solving the exact string searching problem one can derive a variant

```
f'::(string α)→(string α)→num
f' p t = hd (f p t)
```

solving the simplified form of the exact string searching problem. The laziness guarantees the efficiency of the function f': to evaluate the expression $(hd\ (f\ p\ t))$ at most one solution is enumerated.

4.1 The Brute Force Algorithm

The Algorithm for exact string searching that immediately comes to mind is the Brute Force Algorithm:

Algorithm BF Enumerate all m-grams w of t from left to right. Compare p and w character by character in any order. If $p = w$, then report a match. □

Since the character comparisons in the Brute Force Algorithm are done without knowledge about previous comparisons, the algorithm is memoryless.

The Brute Force Algorithm requires a buffer of size $\mathcal{O}(m)$ to hold an m-gram of t. In the worst case, the algorithm needs $\mathcal{O}(m \cdot n)$ time. For example, if $p = a^{m-1}b$ and $t = a^n$ and the comparisons of p and w are performed from left to right, then $m \cdot (n - m + 1)$ character comparisons are necessary. The expected running time of the Brute Force Algorithm is $\mathcal{O}(m + n)$ since the majority of the comparisons between p and an m-gram $w \neq p$ are likely to fail very early (cf. [BM77]).

An optimal functional implementation of the Brute Force Algorithm is given by the function *exsBF*. (The prefix *exs* stands for *exact searching*.)

```
exsBF::(string α)→(string α)→[num]
exsBF p t = [j | (j,s)←zip2 [m..] (suffixes t); p = take m s]
            where m = #p
```

exsBF generates for each $j \geq m$ a lazy list representing the suffix s of t beginning at position $j - m + 1$. It compares p from left to right with the prefix w of s of length m, using the polymorphic built-in predicate "=". If $p = w$, then j is a solution to the exact string searching problem. Note that the expression $(take\ m\ s)$ is evaluated until a mismatch is detected, or p equals $s_1 \ldots s_m$. After p and $(take\ m\ s)$ have been compared, the latter

expression can be garbage collected. The expression ($zip2$ $[m..]$ ($suffixes$ t)) computes only one pair (j, s) at any time. Since s is not evaluated, it takes constant space. Hence, the space consumption of *exsBF* is $\mathcal{O}(m)$. Note that in an imperative implementation of Algorithm BF a considerable part of program code is necessary to handle the buffering of the input string, in order to achieve space efficiency (cf. [BM77]).

There are two general ideas to improve the Brute Force Algorithm:

(1) Record information about previous computations to speed up the comparison of p and an m-gram w of t.

(2) Skip m-grams of t for which it is clear that they do not match p.

In fact, each exact string searching algorithm described in this chapter follows one or both of these ideas:

- The Knuth-Morris-Pratt Algorithm described in section 4.2 follows idea (1).
- The Karp-Rabin Algorithm described in section 4.3 and the algorithms of Boyer-Moore type described in section 4.4 follow idea (2).
- The Chang-Lawler Algorithm described in section 4.5 follows both ideas.

Note that we do not consider string searching algorithms that are based on bit-vector operations (see [WM92, BYG92]). This is because these algorithms are limited to short patterns.

4.2 The Knuth-Morris-Pratt Algorithm

Knuth and Pratt discovered an exact string searching algorithm that is much better in its worst case behavior than the Brute Force Algorithm. In parallel to these authors, Morris discovered virtually the same algorithm. They published their algorithm together in [KMP77].

The Knuth-Morris-Pratt Algorithm processes the input string online character by character and compares it with the pattern from left to right. The basic idea is that at each time a mismatch is detected, the "false start" consists of characters that have already been examined. By recording this information, repeating comparisons with the known characters can be avoided.

Every step in the Knuth-Morris-Pratt Algorithm is characterized by the following three items:

- an m-gram w of t which is aligned with p,
- a common prefix s of p and w,
- the actual input character b.

The Knuth-Morris-Pratt Algorithm works as follows:

1. If $|s| = m$, then a match is reported. The longest proper suffix s' of s that is a prefix of p, is determined. p is moved to the right such that s' is aligned with itself.

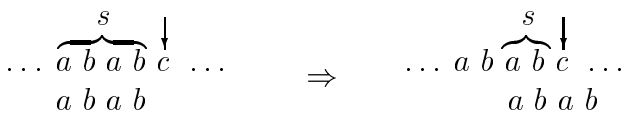
2. If $|s| < m$, then there is a character c in the pattern which is aligned with the actual input character b.

   (a) If $b \neq c$ and $s \neq \varepsilon$, then a mismatch is found. The longest proper suffix s' of s that is a prefix of p, is determined. p is moved to the right such that s' is aligned with itself.

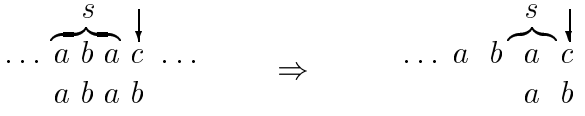
   (b) If $b \neq c$ and $s = \varepsilon$, then a mismatch is found. p is moved one position to the right and the next input character is scanned.

   (c) If $b = c$, then the next input character is read, and the algorithm proceeds with the common prefix sb of p and w.

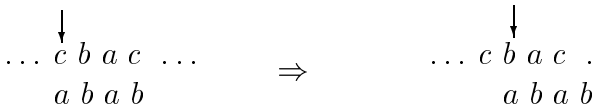
Example 4.2.1 Let $p = abab$. The following alignments exemplify the above cases. The arrow points to the actual input character.

1. 

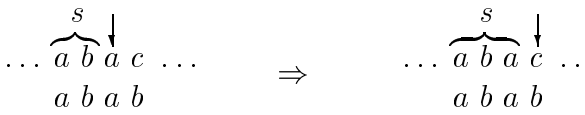
   Since $|s| = m = 4$, a match is reported. ab is the longest proper suffix of s that is a prefix of p. To align ab with itself, the pattern is moved 2 positions to the right.

2. (a) 

   a is the longest proper suffix of s that is prefix of p. To align a with itself, the pattern is moved 2 positions to the right.

   (b) 

   Since $s = \varepsilon$ and $c \neq a$, the pattern is moved one position to the right and the next input character is read.

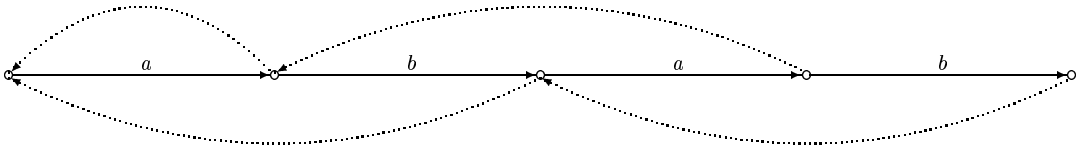
   (c) 

   Since the input character and the corresponding pattern character are equal, the next input character is read and the algorithm proceeds with the common prefix aba of $abac$ and $abab$. □

In every of the four cases described above, the Knuth-Morris-Pratt Algorithm

- never moves the pattern to the left.

Figure 4.1: The KMP-tree for $p = abab$



- either reads the next input character (case (2b) and (2c)), or moves the pattern at least one position to the right (case (1), (2a) and (2b)).

Hence, the four cases can occur at least $2 \cdot n$ times altogether. Cases (2b) and (2c) can easily be accomplished in constant time. To achieve the same efficiency for cases (1) and (2a) we need an additional function $\pi : \{s \mid s \sqsubset p, s \neq \varepsilon\} \to \{s \mid s \sqsubset p, s \neq p\}$ such that $\pi(s)$ is the longest proper suffix of s that is a prefix of p. π only depends on p. Hence, it can be precomputed and stored in a table of size $\mathcal{O}(m)$. In [KMP77], an algorithm is given that computes π in $\mathcal{O}(m)$ time. Thus, the running time of the Knuth-Morris-Pratt Algorithm is $\mathcal{O}(m+n)$ and the space requirement is $\mathcal{O}(m)$.

The worst case performance of the Knuth-Morris-Pratt Algorithm occurs if $p = F_i$ for some $i > 0$ where F_i is the i-th Fibonacci-string, defined as follows [Apo85, Aho90]:

$$F_1 = b, \quad F_2 = a, \quad F_{i+2} = F_{i+1}F_i$$

In [KMP77], it is shown that the maximum number of times the Knuth-Morris-Pratt Algorithm can shift the pattern to the right, while scanning the same input character, is at most $1 + \log_r m$ where $r = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio.

The Knuth-Morris-Pratt Algorithm performs very well for highly repetitive patterns and input strings. Unfortunately, this situation occurs only rarely in actual applications. Thus, in practice the Knuth-Morris-Pratt Algorithm is not likely to be significantly faster than the Brute Force Algorithm. However, there is an important virtue of the Knuth-Morris-Pratt Algorithm. At each time of the computation the algorithm must store only the scanned prefix s of p and the actual input character. Thus, it achieves the space efficiency of $\mathcal{O}(m)$ *without* buffering parts of the input string. This considerably simplifies the implementation.

In section 5.2, we describe how the Knuth-Morris-Pratt Algorithm can be generalized to search for several patterns simultaneously.

4.2.1 Implementation

Most implementations of the Knuth-Morris-Pratt Algorithm represent the function π by an integer array (see [KMP77, CLR90, Aho90, Ste94]). Instead, we represent π by the suffix links of the atomic $\mathcal{A}^+$-tree T such that $words(T) = \{s \in \mathcal{A}^* \mid s \sqsubset p\}$. T is called KMP-tree for p. According to Definition 3.2.4, there is a suffix link $\overline{s} \longrightarrow \overline{v}$ in T if and only if $\pi(s) = v$. Obviously, T has $m+1$ nodes and m edges. Hence, it takes $\mathcal{O}(m)$ space. Figure 4.1 shows the KMP-tree for $p = abab$.

The KMP-tree T for p has a linear structure without branching nodes. It is computed by the function *nobranch* in $\mathcal{O}(m)$ time. Note that the suffix links are left undefined. Moreover,

the node $\overline{p}$ in T is annotated with true and all other nodes with false. To obtain the suffix links for T we will use the function *addlinks* introduced in section 3.7.1.

```
nobranch::(string α)→tree α bool
nobranch (a:u) = N [(a,nobranch u)] undef False
nobranch [] = N [] undef True
```

The Knuth-Morris-Pratt Algorithm is realized as a traversal of the KMP-tree. This plays the role of a memory, in which information about previous character comparisons is recorded. Suppose that w is the m-gram of t, p is aligned with. The common prefix s of w and p (see page 96) is represented by the node $\overline{s}$ of T. If the next input character b matches the next pattern character, then there is an edge $\overline{s} \xrightarrow{b} \overline{sb}$ outgoing from $\overline{s}$. This edge is traversed. Otherwise, the suffix link $\overline{s} \longrightarrow \overline{p_1 \ldots p_i}$ is traversed. This avoids the comparison of $p_1 \ldots p_i$ with itself.

The different cases of the Knuth-Morris-Pratt Algorithm correspond to the different equations defining the function *next* (see section 3.7.1):

- case (2c) corresponds to the first equation,
- case (2b) to the second equation and
- cases (1) and (2a) to the third equation.

Thus, the search phase of the Knuth-Morris-Pratt Algorithm is accomplished by iteratively applying the function *next*. In particular, the expression (*scanl next root t*) returns the list $[\overline{s_0}, \overline{s_1}, \ldots, \overline{s_n}]$ of nodes where $\overline{s_0}$ is the *root* of T and $\overline{s_{j+1}} = next(\overline{s_j}, t_{j+1})$ for each $j, 0 \leq j \leq n-1$ (see section 2.1.4). If $\overline{s_j} = \overline{p}$ for some j, then a match ending at position j is reported.

```
iternext::(tree α bool)→(string α)→[num]
iternext root t = [j | (j,N es link True)←zip2 [0..] (scanl next root t)]
```

The combination of list comprehensions with the higher order function *scanl* will be used in several places throughout the thesis. It provides a very convenient notation to implement the search phase of string searching algorithms which scan the input string character by character, thereby performing state transitions.

The Knuth-Morris-Pratt Algorithm is implemented by the function *exsKMP*.

```
exsKMP::(string α)→(string α)→[num]
exsKMP = iternext.(addlinks setlink).nobranch
```

In a first phase, the linear structure of T including the boolean annotation is computed in $\mathcal{O}(m)$ time. In the second phase, the suffix links are added using the function *addlinks*. According to Theorem 3.7.3, this takes $\mathcal{O}(l \cdot |T|)$ steps, where l is the average number of edges outgoing from the nodes visited during the computation of the suffix links. In our case, $l = 1$ and $|T| = m + 1$. Hence, the function *addlinks* requires $\mathcal{O}(m)$ steps. In the third

phase, which is accomplished by the function *iternext*, at most $2 \cdot n$ suffix links and labeled edges of T are traversed. Each such step takes constant time. Thus, the third phase takes $\mathcal{O}(n)$ time, and our implementation achieves a total running time of $\mathcal{O}(m+n)$ and requires $\mathcal{O}(m)$ space. This is optimal. Note that the function *next* can considerably be optimized if it is applied to a KMP-tree, in which each node either has one or zero outgoing edges.

Several people have noticed that there is a strong resemblance of the preprocessing phase and the search phase of the Knuth-Morris-Pratt Algorithm [Sed88, Aho90, Ste94]. However, van der Woude [vdW89] complained that most implementations of the Knuth-Morris-Pratt Algorithm do *not* justice to this strong resemblance. We react upon this statement: in our implementation the function *next* is used in the preprocessing phase (see *addlinks*) and in the search phase (see *iternext*).

Note that the KMP-tree together with the function *next* can be considered as a space efficient $\mathcal{O}(m)$-representation of a deterministic finite automaton $(\mathcal{S}, \mathcal{F}, s_0, nextstate)$ where $\mathcal{S} = nodes(T)$, $\mathcal{F} = \{\overline{p}\}$, $s_0 = root$, and $nextstate = next$.

4.3 The Karp-Rabin Algorithm

The Karp-Rabin Algorithm [KR87] is very similar to the Brute Force Algorithm. The difference is that the Karp-Rabin Algorithm performs integer comparisons, rather than directly comparing p with each m-gram of t, character by character. Let *fingerprint* be a hash function, that maps each string $w \in \mathcal{A}^m$ to an integer. If $\mathit{fingerprint}(p) \neq \mathit{fingerprint}(w)$, then we know that $p \neq w$. If $\mathit{fingerprint}(p) = \mathit{fingerprint}(w)$, then we must still compare p and w, character by character, to make sure we do not have a mismatch. Since the hash function will be chosen such that a collision is very unlikely to occur, most of the m-grams of t that are not equal to p, are skipped.

The Karp-Rabin Algorithm requires the input alphabet to be known in advance. Therefore, we assume that $l = |\mathcal{A}|$ and a bijective function $encode : \mathcal{A} \to \{0, \ldots, l-1\}$ is given. $encode(b)$ is the code of b. *encode* is generalized to each string w of length m as follows:

$$encode_m(w) = \sum_{i=1}^{m} encode(w_i) \cdot l^{m-i}. \tag{4.1}$$

Thus, an m-gram of t is encoded as an m-digit integer in base l. The hash function *fingerprint* is defined by

$$\mathit{fingerprint}(w) = encode_m(w) \bmod q, \tag{4.2}$$

where q is a large prime. q should be as large as possible such that the computation of $(l+1) \cdot q$ is not causing an arithmetic overflow.

Let bvc be an $(m+1)$-gram of t. The code of vc can be computed from the code of bv by eliminating the "influence" of b and adding the "influence" of c. More precisely, we have $encode_m(vc) = (encode_m(bv) - encode(b) \cdot l^{m-1}) \cdot l + encode(c)$.

A fundamental property of the mod-operation is that it can be applied at any step of the computation. This keeps the numbers we are dealing with small. Thus, we can compute the fingerprint of vc from the fingerprint of bv in constant time according to the following equation (see [Sed88, Aho90, Ste94]):

$$\mathit{fingerprint}(vc) = (l \cdot i + encode(c)) \bmod q \tag{4.3}$$

where $i = (l \cdot q + \mathit{fingerprint}(bv) - \mathit{encode}(b) \cdot k) \bmod q$ and $k = l^{m-1} \bmod q$.

Algorithm KR Compute $\mathit{fingerprint}(t_1 \ldots t_j)$ according to equations (4.1) and (4.2). For each $j, m+1 \leq j \leq n$ compute $\mathit{fingerprint}(t_{j-m+1} \ldots t_j)$ from $\mathit{fingerprint}(t_{j-m} \ldots t_{j-1})$ according to equation (4.3). If $\mathit{fingerprint}(p) = \mathit{fingerprint}(t_{j-m+1} \ldots t_j)$ and $p = t_{j-m+1} \ldots t_j$ for some j, then output j. □

The correctness of Algorithm KR is clear. The algorithm requires a table of size $|\mathcal{A}|$ to represent the bijective function *encode*. Hence, the space consumption is $\mathcal{O}(|\mathcal{A}| + m)$. The running time is $\mathcal{O}(m + n + cc)$ where cc is the number of character comparisons performed. The worst case running time occurs when the fingerprints of the m-grams are always identical with the fingerprint of p. This means that m symbols are to be compared in every step. Thus, the worst case running time is $\mathcal{O}(m \cdot n)$. However, the use of the very large prime q makes it extremely unlikely that a collision occurs. In particular, one can show that the fingerprints in every step would be expected, on the whole, to be unequal (see [KR87]). Under these circumstances, each input character is processed in constant time, giving an $\mathcal{O}(m + n)$ expected running time.

4.3.1 Implementation

The Karp-Rabin Algorithm requires the input alphabet $\mathcal{A}$ to be known in advance. For our functional implementation we therefore assume that $\mathcal{A}$ is given as a pair $(\mathit{characters}, \mathit{encode})$ of type $(\mathit{alphabet}\ \alpha)$ (see section 3.8.1). Let $l = |\mathcal{A}|$ and q be a large prime. If we use Horner's rule (cf. [CLR90]) and furthermore apply the mod-operation at each step of the computation, then we obtain a fingerprint in $\mathcal{O}(m)$ time as follows (see [CLR90, Ste94]):

```
fingerprint::num→num→[num]→num
fingerprint l q codelist = foldl nextcode 0 codelist
                           where nextcode i codeb = (l * i + codeb) mod q
```

Note that the function *fingerprint* is applied to a list of codes, rather than a list of characters. The function *nextfp* is applied to the fingerprint *fpbv* of bv and to the codes of the characters b and c. It returns the fingerprint of vc in constant time according to equation (4.3). The constant $(l \cdot q)$ is added to *fpbv* to ensure that the partial result remains positive.

```
nextfp::num→num→num→num→(num,num)→num
nextfp k l q fpbv (codeb,codec) = (l * i + codec) mod q
                                  where i = (l * q + fpbv - codeb * k) mod q
```

The function *exsKR* implements the Karp-Rabin Algorithm.

```
exsKR::(alphabet α)→num→num→(string α)→(string α)→[num]
exsKR (characters,encode) l q p t
  = [j | (j,s)←jslist; p = take m s]
    where m = #p
          multiples = 1:[(l * i) mod q | i←multiples]
          k = multiples!(m-1)
          codep = map encode p
          codet = map encode t
          fpp = fingerprint l q codep
          firstfp = fingerprint l q (take m codet)
          fplist = scanl (nextfp k l q) firstfp (zip2 codet (drop m codet))
          potmatch = [j | (j,fp)←zip2 [m..] fplist; fpp = fp]
          jslist = [(j,drop (j-j') s) | (j,(j',s))←zip2 potmatch ((m,t):jslist)]
```

Initially, *exsKR* evaluates the constants m and k and encodes the characters of p and t. This takes $\mathcal{O}(m+n)$ time. Recall that k is the constant $(l^{m-1} \bmod q)$. To compute k, the infinite list $multiples = [r_0, r_1, \ldots]$ is constructed where $r_0 = 1$ and $r_{k+1} = (r_k \cdot l) \bmod q$ for each $k \geq 0$. Obviously, $k = r_{m-1}$. The computation of k takes $\mathcal{O}(m)$ steps. Starting with $firstfp = fingerprint(t_1 \ldots t_m)$, and using the function *nextfp*, the list *fplist* of fingerprints of the m-grams of t is constructed in $\mathcal{O}(n)$ steps. Note that the expression $(zip2\ codet\ (drop\ m\ codet))$ returns the list $[(encode(t_1), encode(t_{m+1})), \ldots, (encode(t_{n-m}), encode(t_n))]$. Each fingerprint in *fplist* is compared with the fingerprint *fpp* of p. This gives a list *potmatch* of positions where a potential match occurs. For each j in *potmatch*, the pair (j, s) is computed where s represents the suffix $t_{j-m+1} \ldots t_n$ of t. This yields a list *jslist* in $\mathcal{O}(n)$ time. For each element (j, s) in *jslist* it is checked if p is identical to the prefix of s of length m. This gives the positions of the exact matches of p in t. Altogether, *exsKR* takes $\mathcal{O}(|\mathcal{A}| + m)$ space and $\mathcal{O}(m + n + cc)$ time where cc is the number of character comparisons performed. Thus, our implementation is optimal.

4.4 The Boyer-Moore Algorithm

The basic idea of the Boyer-Moore Algorithm [BM77] is to align the pattern with a subword of t, and to look for a match by comparing the characters in p from right to left with the characters in t. The information gathered in this comparison is used to determine a valid shift, that is, a positive value by which p can be moved to the right without missing an occurrence of p in t.

The Boyer-Moore Algorithm incorporates two heuristics to determine a valid shift. The "good-suffix heuristic" and the "bad-character heuristic"[1]. Both are solely based on the pattern and the input alphabet. When a match or a mismatch occurs, each heuristic proposes a valid shift, and the pattern is moved to the right by the larger of these shifts. The heuristics often allow to skip altogether the examination of many input characters.

In the following two sections, we explain the good-suffix and the bad-character heuristic in detail. Section 4.4.3 considers the efficiency of the Boyer-Moore Algorithm. Sections 4.4.4

[1] The terms good-suffix heuristic and bad-character heuristic are taken from [CLR90].

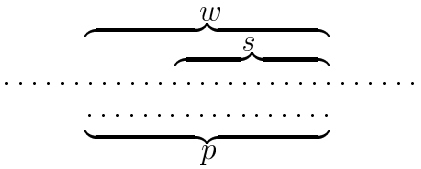
and 4.4.5 are devoted to Horspool's and Sunday's heuristic, respectively. Finally, in section 4.4.6 we give a functional implementation of the described algorithms.

4.4.1 The Good-Suffix Heuristic

Every step of the Boyer-Moore Algorithm, in which the good-suffix heuristic is applied, can be characterized by the following items:

- an m-gram w of t, which is aligned with p,
- the longest common suffix s of p and w.

s is the *good suffix* of w. The following figure shows how the different items are related.



The good-suffix heuristic works as follows:

1. If s is a nested suffix of p, then the heuristic looks for the rightmost nested occurrence of s in p which is not a suffix of p. It proposes a shift that aligns s with this occurrence.
2. If s is not a nested suffix of p, then the good-suffix heuristic looks for the longest proper suffix s' of s which occurs as a prefix of p. It proposes a shift that aligns s' with this occurrence.

Note that the empty string is always a nested suffix. Hence, if $s = \varepsilon$, then the good-suffix heuristic determines the valid shift 1, to align the empty string with its rightmost nested occurrence in p.

Example 4.4.1 Let $p = cdadaad$ and consider the following alignments exemplifying the two cases of the good-suffix heuristic.

1. $$\begin{array}{l} \ldots\ a\ d\ c\ a\ c\ \overbrace{a\ d}^{s}\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ c\ a\ d\ \ldots \\ \qquad\qquad c\ d\ a\ d\ a\ a\ d \end{array}$$

   The good suffix ad of $adcacad$ is a nested suffix of p, and p_3p_4 is the rightmost nested occurrence of ad in p. Hence, we can safely shift the pattern 3 positions to the right, to align ad with p_3p_4.

2. $$\begin{array}{l} \ldots\ a\ d\ c\ a\ \overbrace{a\ a\ d}^{s}\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ a\ a\ d\ \ldots \\ \qquad\qquad\qquad c\ d\ a\ d\ a\ a\ d \end{array}$$

   The good suffix aad of $adcaaad$ is not a nested suffix of p. The empty string is the longest proper suffix of aad that occurs as a prefix of p. Therefore, we can safely shift the pattern 7 positions to the right, to align the empty suffix of aad with the empty prefix of p. □

Definition 4.4.2 The good-suffix heuristic is specified by a function $gsshift : \{s \mid p \int s\} \to \{1, \ldots, m\}$, which is defined as follows:

$$gsshift(s) = \begin{cases} \min\{|u| \mid u \in \mathcal{A}^+, s' \in \mathcal{A}^*, s \int s', p = s'u\}, & \text{if } s \text{ is not nested} \\ \min\{|u| \mid u \in \mathcal{A}^+, p \int su\}, & \text{otherwise} \end{cases}$$

□

Lemma 4.4.3 $gsshift(s)$ is a valid shift.

Proof Obvious. □

Example 4.4.4 If $p = cdadaad$, then $gsshift(ad) = gsshift(d) = 3$, $gsshift(\varepsilon) = 1$, and $gsshift(s) = 7$, for $s \in \{cdadaad, dadaad, adaad, daad, aad\}$. □

The function *gsshift* corresponds to table $delta_2$ in [BM77]. Since *gsshift* depends only on the pattern, it can be preprocessed and stored in a table. The linear time preprocessing algorithms presented in the literature are complicated and difficult to understand (see [Sed88, page 278]). In fact, Knuth's preprocessing algorithm [KMP77] was incorrect. It was corrected by Rytter [Ryt80] and, according to [Smi82], later modified by Mehlhorn (see also [Aho90]).

In a short remark, Crochemore stated that the compact suffix tree of the reverse of p contains all functions precomputed for the Boyer-Moore Algorithm (see [Cro86, page 64]). Unfortunately, he did not explain *how* to obtain these functions from the compact suffix tree. Crochemore's remark motivated us to have a closer look on the relation between suffix trees and the preprocessed functions of the Boyer-Moore Algorithm. We found a simple algorithm, called GST, that uses the compact suffix tree of $p\$$ to construct the function *gsshift*. Algorithm GST is different from the algorithms presented in the literature. It can easily be implemented in our functional framework. The idea is to annotate $T = cst(p\$)$ with some extra information and to collect this information for each suffix of p. In the following, we will explain Algorithm GST in detail.

The non-nested suffixes of p form a contiguous segment of the list of all suffixes of p. The same is true for the nested suffixes. More precisely, a suffix s of p is nested if and only if $|\alpha(p)| \geq |s|$. (Recall that $\alpha(p)$ denotes the longest nested suffix of p.) For this reason, the construction of *gsshift* is accomplished in two phases. In the first phase, *gsshift* is constructed for the non-nested suffixes, in the second for the nested suffixes of p. As usual, a suffix of p is represented by its location in T. Changing from one suffix to the next is done efficiently using the function *linkloc* (see Definition 3.2.14).

The First Phase of Algorithm GST

Suppose s is a non-nested suffix of p. Let s' be the longest proper suffix of s that is a prefix of p. Then $gsshift(s) = m - |s'|$ by Definition 4.4.2. Since s' is a nested suffix of p, it is also a suffix of $\alpha(p)$. Suppose x is another non-nested suffix of p. Then s' is a proper suffix of x as well. Moreover, s' is the longest proper suffix of x that is a prefix of p. Thus, $gsshift(s) = m - |s'| = gsshift(x)$, that is, the *gsshift*-value is the same for all non-nested suffixes of p. Based on this observation, we can compute $gsshift(s)$ for all non-nested suffixes s of p by the following three steps:

(1) Compute $\alpha(p)$.

(2) Compute the longest suffix s' of $\alpha(p)$ that is a prefix of p.

(3) Set $gsshift(s) = m - |s'|$ for all non-nested suffixes s of p.

There are two methods to compute $\alpha(p)$.

- If we construct $cst(p\$)$ with Ukkonen's online algorithm (see section 3.5.2), then we get $\alpha(p)$ as a by-product with the last but one intermediate tree $cst(p)$.
- Since a suffix s of p is nested if and only if $loc_T(s)$ is a node location, we can compute $\alpha(p)$ as the longest suffix of p, that has a node location in T.

In our functional implementation of Algorithm GST (see section 4.4.6), we use the second method since it does not assume a particular suffix tree algorithm. To accomplish step (2), we annotate T.

Definition 4.4.5 The boolean annotation *isprefix* of T is defined as follows: $isprefix(\overline{v})$ is true if and only if v is a prefix of $p\$$. □

Obviously, $isprefix(\overline{v})$ is true if and only if $|v| = m + 1$ or there is an edge $\overline{v} \xrightarrow{u} \overline{vu}$ in T such that $isprefix(\overline{vu})$ is true. Therefore, *isprefix* can be computed in $\mathcal{O}(m)$ time by a single traversal of T.

The Second Phase of Algorithm GST

Suppose s is a nested suffix of p. By Definition 4.4.2 $gsshift(s)$ is the length of the shortest non-empty string u such that su is a suffix of p. Obviously, s is a right-branching subword of $p\$$ since the first character of u is not the sentinel character \$. Hence, there is a branching node $\overline{s}$ in T. In terms of the suffix tree, we have $gsshift(s) = j - 1$ where j is the length of the second shortest path from $\overline{s}$ to a leaf of T. To obtain $gsshift(s)$, we annotate T such that j is available at node $\overline{s}$.

Definition 4.4.6 The annotation $minpathlen : nodes(T) \to \mathcal{P}(\mathbb{N}_0)$ is defined as follows:

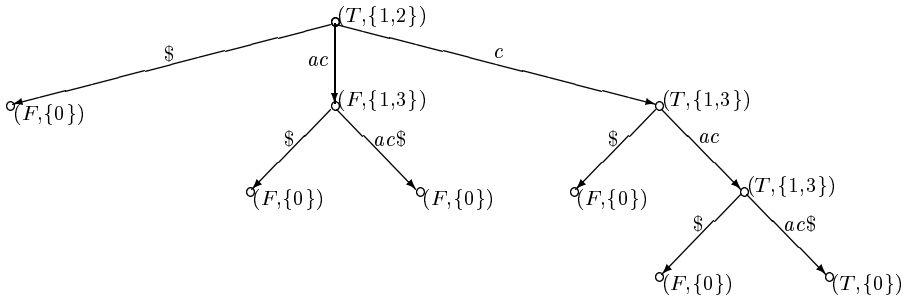
$$minpathlen(\overline{v}) = \begin{cases} \{0\}, & \text{if } \overline{v} \text{ is a leaf} \\ twomin\{|w| \mid w \in \mathcal{A}^+, \overline{vw} \in leaves(T)\}, & \text{otherwise} \end{cases}$$

where $twomin(M) = \{\min(M), \min(M \setminus \{\min(M)\})\}$ for all $M \subseteq \mathbb{N}_0$, $|M| \geq 2$. □

Note that *minpathlen* is welldefined since from each branching node there are at least two non-empty paths of different lengths to some leaves of T.

We have $minpathlen(\overline{v}) = twomin\{|u| + i \mid \overline{v} \xrightarrow{u} \overline{vu} \in edges(T), i \in minpathlen(\overline{vu})\}$ for each branching node $\overline{v}$ in T. Hence, $minpathlen(\overline{v})$ can be computed in time proportional to the number of edges outgoing from $\overline{v}$. Since there are $\mathcal{O}(m)$ edges in T, the annotation *minpathlen* can be obtained in $\mathcal{O}(m)$ time by a single traversal of T.

Figure 4.2: $cst(cacac\$)$ with Annotations *isprefix* and *minpathlen* (*T=True, F=False*)



Note that it does not suffice to store only the length j of the second shortest path at each branching node $\overline{v}$ of T. The length i of the shortest path may be necessary to compute the length of the second shortest path for the father node of $\overline{v}$. If i would not be available at node $\overline{v}$, then it would have to be computed from the nodes below $\overline{v}$. This would lead to a non-linear algorithm.

Example 4.4.7 Let $p = cacac$. The compact suffix tree for $p\$$ with the annotations *isprefix* and *minpathlen* is shown in Figure 4.2. The annotation *minpathlen* at node $\overline{c}$ is computed as follows:

$$\begin{aligned} minpathlen(\overline{c}) &= twomin(\{1+i \mid i \in minpathlen(\overline{c\$})\} \cup \\ &\qquad \{2+i \mid i \in minpathlen(\overline{cac})\}) \\ &= twomin(\{1+0\} \cup \{2+1, 2+3\}) \\ &= \{1,3\} \end{aligned}$$

The value 3 in $minpathlen(\overline{c})$ is obtained from the length 1 of the *shortest* path from $\overline{cac}$ to a leaf. If the value 1 would not be available at $\overline{cac}$, it would have to be computed from the nodes below $\overline{cac}$. □

The second phase of Algorithm GST computes $gsshift(s)$ for each nested suffix s as follows. If $minpathlen(\overline{s}) = \{i, j\}$ and $i < j$, then $gsshift(s) = j - 1$. Altogether, we obtain the following result:

Lemma 4.4.8 Algorithm GST computes *gsshift* in $\mathcal{O}(m)$ space and $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time.

Proof The construction of $cst(p\$)$ with the annotations *isprefix* and *minpathlen* takes $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time. Computing the locations of the suffixes of p takes $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time. Collecting the *gsshift*-value in two phases takes $\mathcal{O}(m)$ time. □

Unlike the $\mathcal{O}(m)$-algorithms in the literature, Algorithm GST has an an additional alphabet factor $|\mathcal{A}_p|$, due to the usage of suffix trees. However, this factor can be eliminated if one uses hash tables to represent the edges of the suffix tree. Anyway, Algorithm GST should be compared to the existing methods.

Example 4.4.9 Let $p = cacac$ and T be the compact suffix tree in Figure 4.2. To construct *gsshift*, Algorithm GST computes $loc_T(p) = (\overline{cac}, ac, \$, \overline{cacac\$})$, $loc_T(acac) = (\overline{ac}, ac, \$, \overline{acac\$})$ and $loc_T(cac) = \overline{cac}$. Hence, $\alpha(p) = cac$. Since $isprefix(\overline{cac})$ is true, a prefix of p is found and the first phase is completed with $gsshift(cacac) = gsshift(acac) = m-3 = 2$. Because cac is nested, $gsshift(cac)$ can immediately be computed from $minpathlen(\overline{cac})$, yielding $gsshift(cac) = 3 - 1 = 2$. The like holds for the suffixes ac, c, and ε. Hence, $gsshift(ac) = gsshift(c) = 3 - 1 = 2$ and $gsshift(\varepsilon) = 2 - 1 = 1$. □

The good-suffix heuristic can be improved if one ensures that after a shift a different character is aligned with the character that caused the mismatch when comparing p and w from right to left. This idea first appeared in [KMP77]. It can be specified by the following function $gsshift'$.

$$gsshift'(s) = \begin{cases} \min\{|u| \mid u \in \mathcal{A}^+, s' \in \mathcal{A}^*, s \sqsupset s', p = s'u\}, & \text{if } s \text{ is not nested} \\ \min(\{m\} \cup P), & \text{otherwise} \end{cases}$$
$$\text{where } P = \{|u| \mid u \in \mathcal{A}^+, p \sqsupset su, m = |su| \text{ or } p_{m-|s|} \neq p_{m-|su|}\}$$

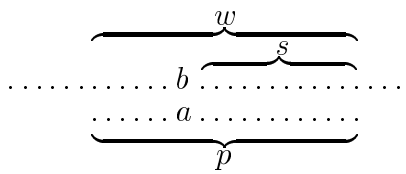
As it only slightly speeds up the running time of the Boyer-Moore Algorithm (see [KMP77]), but complicates preprocessing, we will not further consider this improvement.

4.4.2 The Bad-Character Heuristic

Every step of the Boyer-Moore Algorithm, in which the bad-character heuristic is applied, can be characterized by the following items:

- an m-gram $w \neq p$ of t which is aligned with p,
- the good suffix s of w,
- two different characters b and a such that bs is a suffix of w and as is a suffix of p.

b is the *bad character* of w since it leads to the first mismatch when comparing p and w from right to left. $badpos(w) = m - |s|$ is the position in w where the bad character occurs. Since $|s| < m$, we have $badpos(w) > 0$. The following figure shows how the different items are related:



The bad-character heuristic works as follows:

1. If b does not occur in p, then p may be moved completely past the position in w where b occurs. Hence, $badpos(w)$ is a valid shift.
2. If p_i is the rightmost occurrence of b in p and $badpos(w) > i$, then the pattern can be moved $badpos(w) - i$ positions to the right, before b matches any pattern character. Therefore, $badpos(w) - i$ is a valid shift.

3. If p_i is the rightmost occurrence of b in p and $badpos(w) < i$, then the pattern has to be moved to the left. This recommendation will be ignored by the Boyer-Moore Algorithm since the good-suffix heuristic will propose a shift to the right in all cases.

Example 4.4.10 Suppose $p = cdadaad$ and consider the following alignments exemplifying the three different cases of the bad-character heuristic. The arrow points to the bad character.

1.
$$\begin{array}{l} \ldots\ a\ d\ c\ a\ b\ a\ \underset{\downarrow}{b}\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ b\ a\ b\ \ldots \\ \qquad\qquad\qquad\ c\ d\ a\ d\ a\ a\ d \end{array}$$

The bad character b at position 7 does not occur in p. Therefore, we can safely shift the pattern 7 positions to the right.

2.
$$\begin{array}{l} \ldots\ a\ d\ c\ a\ \underset{\downarrow}{c}\ a\ d\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ c\ a\ d\ \ldots \\ \qquad\qquad\ c\ d\ a\ d\ a\ a\ d \end{array}$$

The bad character c at position 5 occurs in the pattern only to the left of the bad character at position 1. Therefore, we can safely shift the pattern $5 - 1 = 4$ positions to the right.

3.
$$\begin{array}{l} \ldots\ a\ d\ c\ \underset{\downarrow}{a}\ a\ a\ d\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ a\ a\ d\ \ldots \\ c\ d\ a\ d\ a\ a\ d \end{array}$$

The bad character a at position 4 occurs in the pattern to the right of the bad character at position 6. Therefore, the bad-character heuristic proposes a shift of $4 - 6 = -2$ positions.

As can easily be verified, the good-suffix heuristic proposes a shift of $1, 3$ and 7 for the three cases above. Therefore, the Boyer-Moore Algorithm chooses a shift of $7, 4$ and 7. □

The bad-character heuristic uses information about where the bad character occurs in the pattern (if it occurs at all). This information only depends on the input alphabet and the pattern. It is represented by a function $rmostocc_m$.

Definition 4.4.11 For each $j, 1 \leq j \leq m$ the function $rmostocc_j : \mathcal{A} \to \{0, \ldots, j\}$ is defined as follows:

$$rmostocc_j(b) = \begin{cases} \max\{i \mid 1 \leq i \leq j, p_i = b\}, & \text{if } b \text{ occurs in } p_1 \ldots p_j \\ 0, & \text{otherwise} \end{cases}$$

Example 4.4.12 Suppose $p = cdadaad$ and $\mathcal{A} = \{a, b, c, d\}$. Then $rmostocc_m(a) = 6$, $rmostocc_m(b) = 0$, $rmostocc_m(c) = 1$, and $rmostocc_m(d) = 7$. □

Lemma 4.4.13 If $badpos(w) - rmostocc_m(b)$ is positive, then it is a valid shift.

Proof Obvious. □

The function $rmostocc_m$ corresponds to the table $delta_1$ in [BM77]. In particular, we have $rmostocc_m(b) = m - delta_1(b)$ for each $b \in \mathcal{A}$. Note that we have defined several instances of functions containing information of rightmost occurrences. This is because the heuristic of Horspool (see section 4.4.4) requires information about the rightmost occurrence of an input character in $p_1 \dots p_{m-1}$. Such information is given by the function $rmostocc_{m-1}$.

The bad-character heuristic is not applied if a successful match occurs (since then there is no bad character), or if $badpos(w) - rmostocc_m(b) \leq 0$. In both situations, a valid shift is determined solely by the good-suffix heuristic.

An efficient implementation of the bad-character heuristic requires the input alphabet $\mathcal{A}$ to be known in advance. Then the function $rmostocc_j$ can be stored in a table of size $|\mathcal{A}|$. The standard algorithm constructs $rmostocc_j$ in two phases using $\mathcal{O}(|\mathcal{A}| + j)$ steps.

Algorithm RMO [BM77] For each character $b \in \mathcal{A}$ set $rmostocc_j(b) = 0$. For $i = 1, \dots, j$ set $rmostocc_j(p_i) = i$. □

To obtain $rmostocc_m$, we can also use $T = cst(p\$)$ with the annotation $minpathlen$. This is reasonable since T is constructed anyway if we use Algorithm GST for computing $gsshift$.

Algorithm RMOm Set $rmostocc_m(b) = m + 1 - |u| - \min(minpathlen(\overline{bu}))$ for each edge $root \xrightarrow{bu} \overline{bu}$ in T such that $b \neq \$$. Set $rmostocc_m(b) = 0$ for each $b \in \mathcal{A} \setminus \mathcal{A}_p$. □

The correctness of Algorithm RMOm is obvious. The efficiency is $\mathcal{O}(|\mathcal{A}|)$ if we do not count the preprocessing for $cst(p\$)$. In section 4.4.6, we give an implementation of Algorithm RMO and Algorithm RMOm.

4.4.3 Efficiency of the Search Phase

For a space efficient implementation of the search phase of the Boyer-Moore Algorithm, the m-gram of t which is aligned with the pattern, is stored in a buffer of size $\mathcal{O}(m)$. With a little care the buffering operations can be accomplished in $\mathcal{O}(n)$ time. Let cc denote the number of character comparisons performed in the search phase. The running time of the search phase is $\mathcal{O}(n + cc)$.

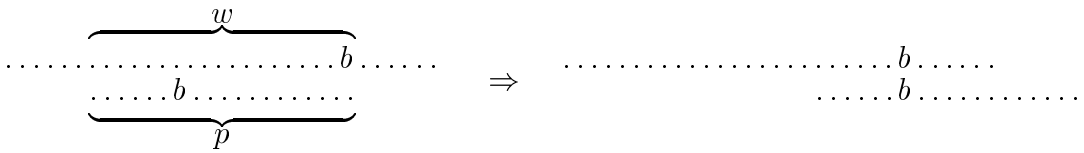
The worst case running time occurs for $p = a^m$ and $t = a^n$. Then $cc = m \cdot (n - m + 1)$ since it takes m character comparisons to verify each of the $n - m + 1$ matches. Therefore, in the worst case the Boyer-Moore Algorithm performs as many character comparisons as the Brute Force Algorithm. In the expected case $cc < n$ (see [BM77]). For this reason, the Boyer-Moore Algorithm is often termed "sublinear". If we consider the simplified exact string searching problem (see page 94), then $cc \in \mathcal{O}(n)$ even in the worst case (see [KMP77]). This occurs for $t = a^n$ and $p = ba^{m-1}$: The good-suffix heuristic proposes a shift of m for every alignment of a^m and p. Hence, there are n/m of these. To verify that a^m does not match p, exactly m character comparisons are required.

The heuristics presented in sections 4.4.1 and 4.4.2 are taken from the paper of Boyer and Moore [BM77]. There are several other heuristics to determine a shift. In the following, we present two simple ones which lead to a very good performance in the expected case.

4.4.4 Horspool's Heuristic

Horspool [Hor80] noted that in the normal usage of the Boyer-Moore Algorithm the good-suffix heuristic does not make much contribution to the overall speed of the search. Its only purpose is to optimize the handling of highly repetitive patterns. Since these are not too common, it is not worthwhile to expend the considerable effort needed to precompute the good-suffix heuristic. Based on this argument, Horspool suggested to do without the good-suffix heuristic.

Let w be the m-gram of t which is aligned with p. To avoid the situation where the bad-character heuristic proposes a backward shift, Horspool's heuristic determines a shift that aligns the input character $b = w_m$ with its rightmost occurrence in $p_1 \ldots p_{m-1}$ (if there is any). This is illustrated by the following figure:



Lemma 4.4.14 $m - rmostocc_{m-1}(w_m)$ is a valid shift.

Proof Obvious. □

Example 4.4.15 Suppose $p = cdadaad$ and $\mathcal{A} = \{a, b, c, d\}$. Then $rmostocc_{m-1}(a) = 6$, $rmostocc_{m-1}(b) = 0$, $rmostocc_{m-1}(c) = 1$, and $rmostocc_{m-1}(d) = 4$. Consider the following alignments where the arrow points to the last character in w.

1.
$$\begin{array}{l} \ldots\ a\ d\ c\ a\ b\ a\ \underset{\downarrow}{b}\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ b\ a\ b\ \ldots \\ \qquad\qquad\qquad\ c\ d\ a\ d\ a\ a\ d \end{array}$$

   We can safely shift p by $7 - rmostocc_{m-1}(b) = 7 - 0 = 7$ positions to the right. The same shift is determined by the Boyer-Moore Algorithm.

2.
$$\begin{array}{l} \ldots\ a\ d\ c\ a\ c\ a\ \underset{\downarrow}{d}\ \ldots \\ \quad\ c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ c\ a\ d\ \ldots \\ \qquad\qquad\ c\ d\ a\ d\ a\ a\ d \end{array}$$

   We can safely shift p by $m - rmostocc_{m-1}(d) = 7 - 4 = 3$ positions to the right. In the same situation, the Boyer-Moore Algorithm shifts the pattern 4 positions to the right. □

The shift proposed by Horspool's heuristic is never larger than the shift determined in the Boyer-Moore Algorithm. This result is not surprising. However, to our knowledge it is not stated elsewhere. Therefore, we give a proof for it:

Lemma 4.4.16 Let $w \in \mathcal{A}^m$. If $w = p$, then $gsshift(w) \geq m - rmostocc_{m-1}(w_m)$. If $w \neq p$, then $\max\{gsshift(s), badpos(w) - rmostocc_m(b)\} \geq m - rmostocc_{m-1}(w_m)$, where s is the good suffix and b is the bad character of w.

Proof Let $w = p$. Then w is the good suffix of w. Since w is not nested, we get $gsshift(w) = m - |s'|$ where s' is the longest proper suffix of w that is a prefix of p. If $s' = \varepsilon$, then $gsshift(w) = m \geq m - rmostocc_{m-1}(w_m)$. If $s' \neq \varepsilon$, then $w_m = p_{|s'|}$. Since $rmostocc_{m-1}(w_m) \geq |s'|$, we get $gsshift(w) = m - |s'| \geq m - rmostocc_{m-1}(w_m)$.

Let $w \neq p$. Suppose s is the good suffix and b is the bad character of w. If $s = \varepsilon$, then $w_m = b \neq p_m$. Hence, $rmostocc_m(b) = rmostocc_{m-1}(w_m) < m$ and therefore

$$\begin{aligned} \max\{gsshift(\varepsilon), badpos(w) - rmostocc_m(b)\} &= \max\{1, m - rmostocc_m(b)\} \\ &= m - rmostocc_m(b) \\ &= m - rmostocc_{m-1}(w_m). \end{aligned}$$

Let $s \neq \varepsilon$ and u be the shortest non-empty string such that $p \sqsupset su$ or $p = s'u$ for a suffix s' of s (cf. Definition 4.4.2). Since w_m is the last character of s and also of s' whenever $s' \neq \varepsilon$, we can conclude $gsshift(s) = |u| \geq m - rmostocc_{m-1}(w_m)$. □

Horspool's heuristic leads to an algorithm which does not have to compute the good suffix. Hence, it can compare the characters of p and w in any order, as specified in Horspool's paper [Hor80]. Most authors ignore that Horspool's heuristic leads to a free choice for the comparison order. They give a specification of the algorithm in which characters are compared from right to left [Aho90, Ste94] like it is necessary in the Boyer-Moore Algorithm. In fact, Baeza-Yates [BY89c] shows that the running time is independent of the comparison order. Thus, let the Boyer-Moore-Horspool Algorithm be the variation of the Boyer-Moore Algorithm that uses Horspool's heuristic and an arbitrary but fixed comparison order.

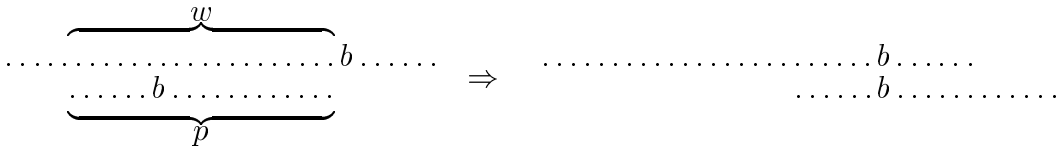
The Boyer-Moore-Horspool Algorithm only needs the function $rmostocc_{m-1}$ to be preprocessed. It can be obtained in $\mathcal{O}(|\mathcal{A}|+m)$ steps by Algorithm RMO. Like for the Boyer-Moore Algorithm a buffering scheme is necessary to store the actual m-gram of t. Therefore, the running time of the Boyer-Moore-Horspool Algorithm is $\mathcal{O}(|\mathcal{A}| + m + n + cc)$ where cc is the number of character comparisons performed. Assume that the comparison order for the Boyer-Moore-Horspool Algorithm specifies j as the last index of p and w to perform the character comparison at. In the worst case, $cc \in \mathcal{O}(n \cdot m)$ (take $t = a^n$ and p such that $p_j = b$ and $p_i = a$, $1 \leq i \leq m$, $i \neq j$). Notice that this is also the worst case if we consider the simplified exact string searching problem.

According to [BYR92], $cc \in \mathcal{O}(n)$ in the expected case. Thus, one can expect that the Boyer-Moore Algorithm is faster than the Boyer-Moore-Horspool Algorithm. However, doing without the good-suffix heuristic, the preprocessing for the Boyer-Moore-Horspool Algorithm is simpler and faster, although it is still $\mathcal{O}(|\mathcal{A}| + m)$. In fact, empirical measurements by Baeza-Yates [BY89c] show that Horspool's variant, with a right to left comparison order, outperforms the Boyer-Moore Algorithm for almost all pattern lengths and alphabet sizes, despite the result of Lemma 4.4.16.

Due to the free choice of the comparison order, the Boyer-Moore-Horspool Algorithm can be combined with any other linear time exact string searching algorithm in order to improve the average case behavior. Baeza-Yates [BY89a, BY89b], for instance, obtains a hybrid algorithm that combines the Boyer-Moore-Horspool Algorithm with the Knuth-Morris-Pratt Algorithm. In practice, the combination is slightly faster than the Boyer-Moore-Horspool Algorithm for searches in English text.

4.4.5 Sunday's Heuristic

The heuristics applied in the Boyer-Moore Algorithm as well as Horspool's heuristic always shift the pattern between 1 and m positions to the right. Hence, the new input character b immediately to the right of w is a part of the next m-gram, p will be aligned with. Based on this observation, Sunday [Sun90] suggested to shift the pattern to the right such that b is aligned with its rightmost occurrence in p. If b does not occur in p, then p can safely be shifted $m+1$ positions to the right. The following picture illustrates Sunday's heuristic.



Lemma 4.4.17 If w ends at position $j < n$ in t, then $1 + m - rmostocc_m(t_{j+1})$ is a valid shift.

Proof Obvious. □

Example 4.4.18 Let $p = cdadaad$. Consider the following alignments where the arrow points to the input character that determines the shift.

1.

$$\begin{array}{l} \ldots\ a\ d\ c\ a\ b\ a\ b\ \underset{\downarrow}{d}\ \ldots \\ \quad c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ b\ a\ b\ d\ \ldots \\ \quad\quad\quad c\ d\ a\ d\ a\ a\ d \end{array}$$

We can safely shift p by $1 + m - rmostocc_m(d) = 1 + m - m = 1$ position to the right. Notice that Horspool's heuristic determines a shift of 7 in the same situation.

2.

$$\begin{array}{l} \ldots\ a\ d\ c\ a\ c\ a\ d\ \underset{\downarrow}{b}\ \ldots \\ \quad c\ d\ a\ d\ a\ a\ d \end{array} \quad \Rightarrow \quad \begin{array}{l} \ldots\ a\ d\ c\ a\ c\ a\ d\ b\ \ldots \\ \quad\quad\quad\quad\quad\quad\quad\quad c\ d\ a\ d\ a\ a\ d \end{array}$$

We can safely shift p by $1 + m - rmostocc_m(b) = 1 + 7 - 0 = 8$ positions to the right. In the same situation, Horspool's heuristic proposes a shift of 3. □

This example shows that neither Sunday's nor Horspool's heuristic should always be preferred. Consequently, Smith [Smi91] suggested to combine both heuristics, by taking the maximum of the shifts proposed by the different heuristics. We conjecture that this only leads to a small improvement of the running time.

Sundays heuristic leads to a string searching algorithm that is similar to the Boyer-Moore-Horspool Algorithm. Therefore, the remarks on the efficiency of the Boyer-Moore-Horspool Algorithm (see section 4.4.4) hold for this new algorithm as well.

Sunday [Sun90] describes three different string searching algorithms, all of which use his heuristic, but differ in the way the symbol comparisons are performed. The simplest of the three is the Quick Search Algorithm, which performs the character comparisons from left to right. The other two methods try to precompute an optimal comparison order which must

be represented by an additional data structure. The Maximal Shift Algorithm chooses a comparison order such that in case of a mismatch, the distance to the next possible pattern position is maximized. This is done by sorting the pattern characters in descending order of their distance to their next leftward occurrence in the pattern, or, if there is none, to the position where the pattern begins.

In the Optimal Mismatch Algorithm, the probability of an early detection of a mismatch is maximized by comparing rarer symbols first. This requires the pattern symbols to be sorted in descending order of their frequency of occurrence in the input string. Hence, knowledge about the character distribution in the input string is needed in advance. Smith [Smi91] has developed an adaptive version of the Optimal Mismatch Algorithm with the advantage that it does not require such knowledge. Improvements involving character distributions were also suggested by Baeza-Yates [BY89b, BY89a] to be used in the Boyer-Moore-Horspool Algorithm. They can obviously be applied to the Brute Force Algorithm as well.

Based on empirical data, Sunday found the Optimal Mismatch Algorithm to be the fastest of his methods, and reported all three to be superior in average case performance to the Boyer-Moore Algorithm. A speed advantage of 10-20 percent is, for instance, reported for the Optimal Mismatch Algorithm. The speed differentials, however, decrease with an increase in the pattern length [Ste94].

Sunday's algorithms have also been empirically compared to a simplified Boyer-Moore variant by Pirklbauer [Pir92]. In this study, it was found that for long strings and for the case where preprocessing time was taken into account, the Maximal Shift and Optimal Mismatch Algorithm behaved rather badly. Their preprocessing overheads were found to be significant, especially when searching only for the first occurrence of the pattern. The Quick Search Algorithm was recommended because of its ease of implementation and of its performance which was similar to that of the Boyer-Moore Algorithm. Unfortunately, neither Sunday nor Pirklbauer considered the Boyer-Moore-Horspool Algorithm. We conjecture, that the latter performs similar to the Quick Search Algorithm, due to the strong resemblance of both algorithms.

Before we consider implementation issues, we briefly outline some ideas for improving the algorithms considered in this section. For the algorithms of Boyer-Moore type, the average shift value grows with larger input alphabets. Improved performance can therefore be obtained by increasing the size of the input alphabet. This technique has been considered in [KMP77] and in [Sch88]. Baeza-Yates [BY89b] proposed a simple alphabet transformation which leads to a practical implementation of a Boyer-Moore-Horspool variant. Essentially this involves grouping k symbols of the pattern as a "supersymbol". This increases the size of the input alphabet to $|\mathcal{A}|^k$ and has been shown to be practical if $\mathcal{A}$ is small.

The Boyer-Moore Algorithm verifies a match or a mismatch between the actual m-gram of t and p, without memorizing any information about previous character comparisons. For instance, if a mismatch between w_{m-1} and p_{m-1} occurs, and if the Boyer-Moore Algorithm shifts p one position to the right, then the information that $w_m = p_m$ is completely lost. Knuth [KMP77] recognized this fact. He suggested a variation of the Boyer-Moore Algorithm which uses a deterministic finite automaton. Each state of this automaton carries partial information about the characters of the actual m-gram of t that match the corresponding characters of p. Unfortunately, the number of states can be 2^m in the worst case. Recently, Baeza-Yates et al. [BYCG94] have formalized Knuth's idea. Several improvements for both

the average and the worst case behavior of so-called Boyer-Moore automata are proposed. In particular, it is shown that for a certain class of patterns the Boyer-Moore automata have $\mathcal{O}(m^3)$ states if $\mathcal{A}$ is the binary alphabet.

Crochemore et al. [CCG+94] presented an improved Boyer-Moore Algorithm, called TurboBM. If one considers the simplified string searching problem, then TurboBM performs at most $2 \cdot n$ character comparisons. Note that the upper bound for the Boyer-Moore Algorithm is $3 \cdot n$, as shown in [Col90]. The idea of TurboBM is to memorize the previous good suffix when comparing the actual m-gram of t with p. On the one hand, this often allows to skip several character comparisons. On the other hand, it often allows to compute a shift value which is larger than the shift value proposed by the good-suffix and the bad-character heuristic. Besides TurboBM, Crochemore et al. [CCG+94] also devised variations of the Boyer-Moore Algorithm, called reverse factor algorithms. These algorithms scan the m-gram w of t from right to left, as long as the scanned suffix of w is a subword of p. Determining the shift value from this suffix, (usually) leads to a larger shift value and an improved average case performance. In particular, if $|\mathcal{A}| \geq 2$, then the average number of character comparisons is $\mathcal{O}(n \cdot (\log_{|\mathcal{A}|} m)/m)$.

4.4.6 Implementation

The Preprocessing

Let $T = cst(p\$)$. For each node $\overline{v}$ in T, the annotations $isprefix(\overline{v})$ and $minpathlen(\overline{v})$ are represented by a pair consisting of a boolean value and an ordered list of at most two elements. Let $\overline{v}$ be an inner node in T such that es represents the edges outgoing from $\overline{v}$. The expression $(gettwomin\ es)$ returns $twomin(\overline{v})$ in time proportional to the length of es. This is accomplished as follows. For each edge $\overline{v} \xrightarrow{u} \overline{vu}$ outgoing from $\overline{v}$ the list $[l + i \mid i \leftarrow minpathlen(\overline{vu})]$, where $l = |u|$ is computed. The resulting lists are ordered and merging them gives an ordered list, of which the first two elements are taken.

```
gettwomin::[cedge α (β,[num])]→[num]
gettwomin = take 2.foldl merge [].map collect
            where collect ((u,l),N es link (b,minpathlen')) = [l+i | i←minpathlen']
```

The function *isprefixminpathlen* computes the annotations *isprefix* and *minpathlen* as described in section 4.4.1. This takes $\mathcal{O}(m)$ steps for all nodes of T.

```
isprefixminpathlen::num→annotationfunction α (bool,[num])
isprefixminpathlen m d es
  = (m+1 = d,[0]),                                                    if es = []
  = ([0 | (s,N es link (True,minpathlen))←es] ~= [],gettwomin es), otherwise
```

The list $gsshiftlist = [gsshift(p_1 \ldots p_m), gsshift(p_2 \ldots p_m), \ldots, gsshift(p_m), gsshift(\varepsilon)]$ represents *gsshift*. It is computed in two phases by the function *gst* which implements Algorithm GST.

```
gst::(string α,num)→(ctree α (bool,[num]))→[num]
gst p root = collectshifts 0 (suffixlocs (:[]) (getloc root p))
```

The second argument of *gst* is the *root* of $cst(p\$)$. *gst* computes the locations of all suffixes of p. This is accomplished in $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ steps by applying the function (*suffixlocs* (: [])) (see section 3.4.6) to $loc_T(p)$. Then *gst* calls the function *collectshifts* to obtain the shifts.

```
collectshifts::num→[location α (bool,[num])]→[num]
collectshifts nonnested (LocE node v w node':slocs)
  = collectshifts (nonnested+1) slocs
collectshifts nonnested slocs
  = take nonnested (repeat i) ++ [j-1 | LocN (N es link (isprefix,[i,j]))←slocs]
    where i = hd [i | (i,LocN (N es link (True,mpl)))←zip2 [nonnested..] slocs]
```

collectshifts has two arguments: The number *nonnested* of previous non-nested suffixes of p, and a list of locations for the remaining suffixes. If an edge location is encountered, then *nonnested* is incremented. If a node location is encountered, then we have reached the location of $\alpha(p)$ and *collectshifts* computes a value $i = m - |s'| = nonnested + |\alpha(p)| - |s'|$ where s' is the longest suffix of $\alpha(p)$ that is a prefix of p. Obviously, $gsshift(s) = i$ for each non-nested suffix s of p. Hence, the first part of *gsshiftlist* is returned by the expression (*take nonnested* (*repeat i*)) The implementation of the second phase, which yields the list of *gsshift*-values for the nested suffixes of p, is straightforward. The remaining list *slocs* of node locations is scanned. At each node $\overline{s}$ a list $[i, j]$ is found which represents $minpathlen(\overline{s})$. Obviously, $j - 1$ is the correct value of $gsshift(s)$. The running time for *gst* is $\mathcal{O}(|\mathcal{A}_p| \cdot m)$.

We assume that the input alphabet $\mathcal{A}$ is given as a pair $(characters, encode)$ of type (*alphabet* α). The function $rmostocc_m$ is represented by an array *rmoarraym* of size $|\mathcal{A}|$ such that (*lookup* (*encode b*) *rmoarraym* $= rmostocc_m(b)$) for each $b \in \mathcal{A}$. The function *makermoarraym* implements Algorithm RMOm to return *rmoarraym*. Note that it requires extension 1.

```
makermoarraym::[α]→α→(string α,num)→(ctree α (bool,[num]))→array num
makermoarraym characters sentinel (p,m) root
  = makearray l (mergealpha characters occ)
    where l = #characters
          occ = [(c,m+1-j-hd (snd (seltag node))) |
                 ((c:u,j+1),node)←seledges root; sentinel ~= c]
```

The fourth argument of *makermoarraym* is the *root* of T. The entries for the characters occurring in p are obtained from the *minpathlen*-annotation of the nodes immediately below the *root* (see Algorithm RMOm). This yields a list *occ* in $\mathcal{O}(|\mathcal{A}_p|)$ steps. Using a function *mergealpha*, the list *occ* is merged with the zero-values for the characters of $\mathcal{A}$ *not* occurring in p. The resulting list is transformed into an array. Thus, *makermoarraym* takes $\mathcal{O}(|\mathcal{A}|)$ time and space to return *rmoarraym*.

```
mergealpha::[α]→[(α,num)]→[num]
mergealpha (a:as) ((c,j):cjs) = 0:mergealpha as ((c,j):cjs), if a < c
                              = j:mergealpha (a:as) cjs,     if c < a
                              = j:mergealpha as cjs,         otherwise
mergealpha as [] = [0 | a←as]
mergealpha [] cjs = map snd cjs
```

For the algorithms using Horspool's or Sunday's heuristic we implement a function *makermoarray* which computes for each $j, 1 \leq j \leq m$ an array *rmoarray* such that for each $b \in \mathcal{A}$ we have $(lookup\ (encode\ b)\ rmoarray = rmostocc_j(b))$.

```
makermoarray::(alphabet α)→num→(string α)→array num
makermoarray (characters,encode) j p
  = foldl update' (makearray l (take l (repeat 0))) (zip2 (map encode p) [1..j])
    where l = #characters
          update' array (codea,i) = update codea i array
```

makermoarray implements Algorithm RMO. This requires extension 2. Due to the simplicity of Algorithm RMO, our implementation does without the monad of array transformers (see section 2.3.2). The first phase of Algorithm RMO is accomplished by creating an array of size $l = |\mathcal{A}|$. Each of the l entries of the array is initialized with 0. Then the list $[(encode(p_1), 1), \ldots, (encode(p_j), j)]$ is generated. Using the function *foldl*, it is scanned from left to right. For each list element $(codea, i)$ the function *update'* is called which updates the actual array with value i at index *codea*. Note that *foldl* forces *update'* to evaluate its first argument. Moreover, there is no call to the function *lookup* which can duplicate the array. Hence, the array is single threaded and it is save to implement the array updates by overwriting. The running time of *makermoarray* is $\mathcal{O}(|\mathcal{A}| + j)$.

The Boyer-Moore Algorithm

Every step of the Boyer-Moore Algorithm is characterized by a suffix wy of t such that w is aligned with p. To determine the valid shift, the good suffix s of w must be computed in $\mathcal{O}(|s|)$ steps. This is usually accomplished by representing w and p as arrays which are compared from right to left. Equivalently, one can compare w^{-1} and p^{-1} from left to right. So do we.

Definition 4.4.19 A suffix wy of t such that $|w| \leq m$ is represented by a pair (x, y) of strings where $x = w^{-1}$. x is the buffer for t. □

Note that such a pair (x, y) takes $\mathcal{O}(m)$ space since $|x| \leq m$ and y is a lazy list which requires constant space.

Let $1 \leq i \leq m$. Shifting the pattern i positions to the right means to move the first i characters from the remaining suffix to the front of the buffer and to forget the last i characters of the buffer. In other words, a pair (x, y) is transformed into $(y_i y_{i-1} \ldots y_1 x_1 \ldots x_{m-i}, y_{i+1} y_{i+2} \ldots)$. This is accomplished by the function *rightshift*.

```
rightshift::num→num→(string α)→(string α)→(string α,string α)
rightshift m i x y = move i (take (m-i) x) y


move::num→(string α)→(string α)→(string α,string α)
move (i+1) x (b:y) = move i (b:x) y
move i x y = (x,y)
```

The function *exsBM* implements the Boyer-Moore Algorithm.

```
exsBM::(alphabet α)→α→(string α)→(string α)→[num]
exsBM (characters,encode) sentinel p t
  = [j | (j,True)←bmsearch m (reverse (take m t),drop m t)]
    where m = #p
          root = cst (p++[sentinel],m+1) (isprefixminpathlen m)
          rmoarraym = makermoarraym characters sentinel (p,m) root
          gsshiftlist = reverse (gst (p,m) root)
          p' = reverse p
          bmsearch j (x,[]) = [(j,p' = x)]
          bmsearch j (x,y)
            = (j,match):bmsearch (j+shift) (rightshift m shift x y)
              where (match,shift) = bmcmp p' [m,m-1..] gsshiftlist x
          bmcmp (a:u) (i:is) (goodsuffixshift:gssl) (b:v)
            = bmcmp u is gssl v,                                  if a = b
            = (False,max2 goodsuffixshift badcharactershift), otherwise
              where badcharactershift = i-lookup (encode b) rmoarraym
          bmcmp u is [goodsuffixshift] v = (True,goodsuffixshift)
```

exsBM first computes $cst(p\$)$ with the annotations *isprefix* and *minpathlen*. Then it constructs the array *rmoarraym* and the list *gsshiftlist* in reverse order, using the preprocessing functions *makermoarraym* and *gst*. Thus, *exsBM* requires extension 1. The search process is accomplished by a function *bmsearch* which is called with some arguments j and $(x, y) = (w^{-1}, t_{j+1} \dots t_n)$ where $w = t_{j-m+1} \dots t_j$. Initially, $j = m$. *bmsearch* returns a list of pairs $(j, match)$ such that *match* is true if and only if p ends at position j in t. The comparison of $p' = p^{-1}$ and $x = w^{-1}$ is accomplished by a function *bmcmp*. Suppose the arguments of *bmcmp* are of the form

$$(a : u) \;\; (i : is) \;\; (goodsuffixshift : gssl) \;\; (b : v)$$

Then $a = p_i$, $b = w_i$ and $goodsuffixshift = gsshift(w_{i+1} \dots w_m)$. If $a = b$, then *bmcmp* proceeds by comparing $u = p_{i-1} \dots p_1$ and $v = w_{i-1} \dots w_1$ from left to right. If $a \neq b$, then a mismatch is detected. Moreover, b is the bad character of w, $i = badpos(w)$, and $w_{i+1} \dots w_m$ is the good suffix of w. Therefore, by Lemma 4.4.13 and Lemma 4.4.3, the maximum of *goodsuffixshift* and $i - rmostocc_m(b)$ is a valid shift.

Suppose the arguments of *bmcmp* are of the form

$$u \;\; is \; [goodsuffixshift] \;\; v$$

Then $u = \varepsilon$, $v = \varepsilon$, and $p = w$. Hence, p is the good suffix of w and $goodsuffixshift = gsshift(p)$. A match is reported and the shift is determined solely by the good-suffix heuristic.

The preprocessing phase in *exsBM* requires $\mathcal{O}(|\mathcal{A}_p| \cdot m + |\mathcal{A}|)$ time. Now consider the search phase. There are $\mathcal{O}(n)$ calls to the function *move*, each of which moves a different input character to the buffer for t. Due to the laziness, the expression $(take \; (m - i) \; x)$ in the function *rightshift* is evaluated only if a character comparison occurs. Hence, the total running time of the function *take* is $\mathcal{O}(cc)$ where cc is the number of character comparisons performed. Moreover, there are $\mathcal{O}(cc)$ calls to the function *bmcmp*, each of which takes constant time. Thus, we can conclude that the total running time of *bmsearch* is $\mathcal{O}(n + cc)$. Hence, our implementation is optimal.

The Boyer-Moore-Horspool Algorithm

The function *exsBMH* implements the Boyer-Moore-Horspool Algorithm. It compares the characters of the pattern and the input string from right to left.

```
exsBMH::(alphabet α)→(string α)→(string α)→[num]
exsBMH (characters,encode) p t
  = [j | (j,True)←bmh m (reverse (take m t),drop m t)]
    where m = #p
          rmoarray = makermoarray (characters,encode) (m-1) p
          p' = reverse p
          bmh j (x,[]) = [(j,p' = x)]
          bmh j (x,y) = (j,p' = x):bmh (j+shift) (rightshift m shift x y)
                        where shift = m-lookup (encode (hd x)) rmoarray
```

The preprocessing is accomplished by the function *makermoarray*. Hence, *exsBMH* requires extension 2. For the search phase we use a function *bmh* which returns a list of pairs $(j, match)$ such that *match* is true if and only if p ends in t at position j. As in *exsBM*, we represent a suffix wy of t by the pair (x, y), where $x = w^{-1}$ (see Definition 4.4.19). This implies that $w_m = hd\ x$. A valid shift is computed in constant time according to Lemma 4.4.14. To check if w and p match, we compare p^{-1} and x using the built-in predicate "=". Note that "=" compares the characters of the strings from left to right. If a mismatch is found, the comparison is stopped. The preprocessing takes $\mathcal{O}(|\mathcal{A}| + m)$ time. Let cc be the number of character comparisons performed. Similar to *bmsearch* one shows that *bmh* has a running time of $\mathcal{O}(n + cc)$. Hence, our implementation is optimal.

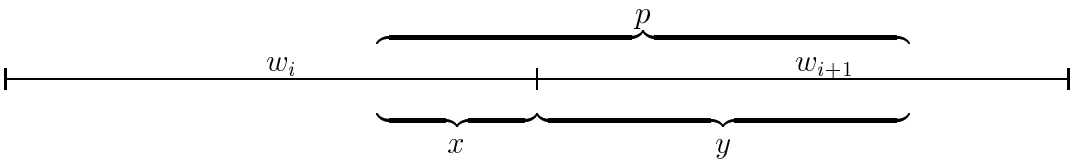
An Algorithm with Sunday's Heuristic

The function *exsBMS* implements an exact string searching algorithm that uses Sunday's heuristic. The algorithm is like the Quick Search Algorithm, except that it compares the characters of p and t from right to left.

```
exsBMS::(alphabet α)→(string α)→(string α)→[num]
exsBMS (characters,encode) p t
  = [j | (j,True)←bms m (reverse (take m t),drop m t)]
    where m = #p
          rmoarray = makermoarray (characters,encode) m p
          p' = reverse p
          bms j (x,[]) = [(j,p' = x)]
          bms j (x,b:y) = (j,p' = x):bms (j+shift) newpair
                          where shift = 1+m-lookup (encode b) rmoarray
                                newpair = (reverse (take m y),drop m y), if 1+m = shift
                                        = rightshift m shift x (b:y),    otherwise
```

exsBMS is very similar to *exsBMH*. The main difference is that in *exsBMS* a shift is determined from the character t_{j+1} (see Lemma 4.4.17) which can easily be accessed since it is the first character b of the remaining suffix $(b : y)$ of the input string. Note that if the shift

Figure 4.3: The Idea of Algorithm ECL



is $m+1$, we do not apply the function *rightshift* to x and $(b:y)$ since this would move $m+1$ characters from $(b:y)$ into the buffer. Instead, we take the prefix of y of length m, reverse it and copy it into the buffer. Moreover, instead of dropping the first $m+1$ characters of $(b:y)$ we drop the first m characters of y to obtain the new remaining suffix of t.

The preprocessing in *exsBMS* takes $\mathcal{O}(|\mathcal{A}|+m)$ time and requires extension 2. The search phase which is accomplished by the function *bms*, takes $\mathcal{O}(n+cc)$ time, where cc is the number of character comparisons performed. Hence, our implementation is optimal.

4.5 The Chang-Lawler Algorithm

An algorithm of Chang and Lawler [CL90] solves the exact string searching problem by traversing the compact suffix tree of $p\$$. On the one hand, the algorithm avoids repeating certain successful character comparisons, similar to the Knuth-Morris-Pratt Algorithm. On the other hand, the algorithm skips over parts of the input string, similar to the Boyer-Moore Algorithm, but without knowing the input alphabet in advance. Thus, the algorithm of Chang and Lawler combines the virtues of two classical algorithms. In particular, it achieves a worst case running time of $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ and performs less than n character comparisons in the expected case.

The algorithm of Chang and Lawler is widely unknown. We do not know any reference to it, except for a short remark in [CL90]. Even the successor paper [CL94] did not mention the algorithm. Since we think that it is very interesting, we take some time to give the details omitted in [CL90]. Moreover, we describe a variant of the algorithm, suggested by Robert Giegerich [Gie94b]. The idea of Chang and Lawler is to divide the input string into non-overlapping subwords w_1, w_2, and so on, of length m. It is evident that an exact match of p splits into a prefix x and a suffix y such that x is a suffix of w_i and y is a prefix of w_{i+1} for some $i \geq 0$. This is illustrated in Figure 4.3.

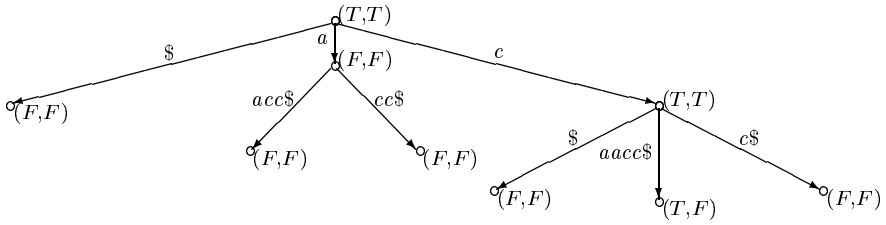
Algorithm ECL [CL90] Let $t = w_0 \ldots w_k$ such that $|w_i| = m$ for $0 \leq i \leq k-1$ and $|w_k| < m$. For each $i, 0 \leq i \leq k-1$ compute

$$\begin{aligned} X_i &= \{m - |x| \mid x \in \mathcal{A}^+, w_i \sqsupset x, x \sqsubset p\} \text{ and} \\ Y_i &= \{|y| \mid y \sqsubset w_{i+1}, p \sqsupset y, |y| < m\}. \end{aligned}$$

For each $r \in X_i \cap Y_i$ output $(i+1) \cdot m + r$. □

Note that if n is a multiple of m then $w_k = \varepsilon$. This additional empty string at the end ensures that for each occurrence of p there is a uniquely determined pair (w_i, w_{i+1}) such that p is a subword of $w_i w_{i+1}$ but not of w_{i+1}.

Figure 4.4: $cst(caacc\$)$ with Annotations *isprefix* and *issuffix*



Theorem 4.5.1 Algorithm ECL correctly solves the exact string searching problem.

Proof Let $p = t_{j-m+1} \dots t_j$ for some $j, m \leq j \leq n$. Then there is an $i, 0 \leq i \leq k-1$ such that p is a subword of $w_i w_{i+1}$ but not of w_{i+1}. Let $q = (i+1) \cdot m$, $x = t_{j-m+1} \dots t_q$, and $y = t_{q+1} \dots t_j$. x is a non-empty suffix of w_i and y is a prefix of w_{i+1} of length $< m$. Since $xy = p$, x is a prefix and y is a suffix of p. Hence, $m - |x| \in X_i$ and $|y| \in Y_i$. Obviously, $m - |x| = |y|$. Thus, $|y| \in X_i \cap Y_i$ and the match ending at $j = q + |y|$ will be detected. □

Let $T = cst(p\$)$. To enumerate the sets X_i and Y_i efficiently, Chang and Lawler suggest to use boolean annotations *isprefix* (see Definition 4.4.5) and *issuffix* of T. These allow to decide in constant time for each location $loc_T(v)$ in T whether v is a prefix or a suffix of p.

Definition 4.5.2 The boolean annotation *issuffix* of T is defined as follows: $issuffix(\overline{v})$ is true if and only if v is a suffix of p. □

Example 4.5.3 The compact suffix tree for $caacc\$$ with the annotations *isprefix* and *issuffix* is shown in Figure 4.4. □

The annotation *isprefix* can be computed in $\mathcal{O}(m)$ time (see section 4.4.1). This also holds for the annotation *issuffix*, since for each node $\overline{v}$ in T we have: $issuffix(\overline{v})$ is true if and only if there is a leaf $\overline{v\$}$ in T. Altogether, T including the annotations can be computed in $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time.

Chang and Lawler do not describe how to compute Y_i and X_i using the two annotations. We will do this in the following. Y_i is not empty since ε is a prefix of w_{i+1} and a suffix of p of length $< m$. The elements of Y_i are enumerated in ascending order by following the path for w_{i+1} down from the *root* of T. Let u be the longest prefix of w_{i+1} that occurs in T. Note that $|u| \geq \max(Y_i)$ since all prefixes of w_{i+1} longer than u are not subwords and hence not suffixes of p. Thus, we can stop the enumeration of Y_i once we have reached the location of u in T. In other words, we can skip parts of the input string, similar to the Boyer-Moore Algorithm. Altogether, Y_i can be enumerated in $\mathcal{O}(|\mathcal{A}_p| \cdot |u|)$ time.

The elements of X_i can be enumerated in ascending order by following the path for w_i down from the *root* of T, traversing a suffix link, if we cannot go further down. After w_i has been completely scanned, we have computed the location of the longest suffix v of w_i that is a subword of p. Using suffix links, we obviate to traverse T from the *root* anew for each suffix of w_i that is longer than v. This is a technique known from the Knuth-Morris-Pratt Algorithm. It avoids repeating comparisons between the pattern characters and the

matching input characters in w_i. All suffixes of w_i longer than v are not subwords and hence not prefixes of p. For all non-empty suffixes s of v, we compute the location of s in T using the function *linkloc*. For each location, we check in constant time if it corresponds to a prefix of p. Note that $r \in X_i \cap Y_i$ implies $r \leq \max(Y_i)$. Hence, to construct the intersection of X_i and Y_i, it suffices to enumerate the elements of X_i which are $\leq \max(Y_i)$. Using the technique described above, this requires $\mathcal{O}(|\mathcal{A}_p| \cdot |u|)$ time. Since Y_i and X_i are enumerated in ascending order, their intersection can be computed in $\mathcal{O}(\max(Y_i)) = \mathcal{O}(|u|)$ time.

Like the algorithms of Boyer-Moore type, the algorithm of Chang and Lawler requires a buffer of size $\mathcal{O}(m)$ to store the actual pair (w_i, w_{i+1}). With a little care the buffering operations can be accomplished in $\mathcal{O}(n)$ time. The running time including preprocessing is $\mathcal{O}(|\mathcal{A}_p| \cdot m + n + |\mathcal{A}_p| \cdot l \cdot (n/m))$ where l is the average length of the longest prefix of w_i, $1 \leq i \leq k$, that is a subword of p. In the worst case, $l = m$. For the expected case Chang and Lawler [CL90] show $l = \log_{|\mathcal{A}|} m$. Thus, if one compares Algorithm ECL with the Knuth-Morris-Pratt Algorithm (in the worst case) and with the Boyer-Moore Algorithm (in the expected case), one notices an additional alphabet factor $|\mathcal{A}_p|$ in both cases. However, if we store the edges of T in a hash table, we can get rid of this factor.

Note that, unlike the Boyer-Moore Algorithm, the Chang-Lawler Algorithm does not have to know the input alphabet in advance. Consequently, the term $|\mathcal{A}|$ does not appear in $\mathcal{O}(|\mathcal{A}_p| \cdot m + n + |\mathcal{A}_p| \cdot l \cdot (n/m))$. A very interesting property of Algorithm ECL is that it can be parallelized since all pairs (w_i, w_{i+1}) can be searched independently.

Recently, Robert Giegerich [Gie94b] suggested to compute the set X_i using $T' = cst(p^{-1}\$)$. Let us call this variation Algorithm ECL$'$. First note that x is a suffix of w_i and a prefix of p if and only if x^{-1} is a prefix of w_i^{-1} and a suffix of p^{-1}. Hence, we can construct X_i similar as Y_i. The elements of X_i are enumerated in descending order by following the path for w_i^{-1} down from the *root* of T'. Let u' be the longest prefix of w_i^{-1} that is a subword of p^{-1}. The enumeration can be stopped once the location of u' in T' is reached. Thus, X_i is enumerated in $\mathcal{O}(|\mathcal{A}_p| \cdot |u'|)$ steps. Let l' be the average length of the longest prefix of w_i^{-1}, $0 \leq i \leq k-1$, that is a subword of p^{-1}. The running time of Algorithm ECL$'$ is $\mathcal{O}(|\mathcal{A}_p| \cdot m + n + |\mathcal{A}_p| \cdot (l + l') \cdot (n/m))$. Since $l' \in \mathcal{O}(l)$, this is the same as for Algorithm ECL. Algorithm ECL$'$ is slightly simpler than Algorithm ECL. It does without suffix links and the annotation *isprefix*. Moreover, it constructs X_i and Y_i in virtually the same way. However, it requires to additionally construct T'.

4.5.1 Implementation

Let $T = cst(p\$)$. For each node in T we represent the annotations *depth*, *isprefix*, and *issuffix* by a triple of type $(num, bool, bool)$ which is computed by the function *depthisprefixissuffix* as described in sections 3.7.3, 4.4.6, and 4.5. This takes $\mathcal{O}(m)$ steps altogether.

```
depthisprefixissuffix::num→annotationfunction α (num,bool,bool)
depthisprefixissuffix m d es
  = (d,m+1=d,False), if es = []
  = (d,isp,iss),     otherwise
    where isp = [0 | (s,N es' link' (d',True,iss'))←es] ~= []
          iss = [0 | ((s,1),N [] link' tag')←es] ~= []
```

Let loc be a location in T. ($locdepth\ loc$) returns the depth of loc. If loc is a node, then $|loc| = depth(loc)$. If loc is an edge location of the form $(\overline{u}, v, w, \overline{uvw})$, then $|loc| = depth(\overline{u}) + |v|$.

```
locdepth::(location α (num,β,γ))→num
locdepth (LocN (N es link (h,isp,iss))) = h
locdepth (LocE (N es link (h,isp,iss)) (v,j) w node') = h+j
```

Let $loc = loc_T(s)$ for some $s \in words(T)$. ($locisprefix\ loc$) is true if and only if s is a prefix of p. The latter holds if $isprefix(ceiling(loc))$ is true. ($locissuffix\ loc$) is true if and only if s is a suffix of p. In particular, if loc is a node, then s is a suffix of p if and only if $issuffix(loc)$ is true. If loc is an edge location $(\overline{u}, v, w, \overline{uvw})$, then s is a suffix of p if and only if $|w| = 1$ and $\overline{uvw}$ is a leaf.

```
locisprefix::(location α (β,bool,γ))→bool
locisprefix = second.seltag.ceiling


locissuffix::(location α (β,γ,bool))→bool
locissuffix (LocN (N es link (h,isp,iss))) = iss
locissuffix (LocE node v (w,i) node') = i=1 & isleaf node'
```

Note that each of the functions $locdepth$, $locisprefix$, and $locissuffix$ takes constant time.

Consider the items w_i, w_{i+1}, X_i, and Y_i as determined in Algorithm ECL. For abbreviation we let $w = w_i$, $w' = w_{i+1}$, $X = X_i$, and $Y = Y_i$. We represent the sets X and Y by the lists $inX = [inX_0, \ldots, inX_{m-1}]$ and $inY = [inY_0, \ldots, inY_{\max(Y_i)}]$ of booleans such that inX_r is true if and only if $r \in X$ and inY_r is true if and only if $r \in Y$. This representation considerably simplifies and speeds up the construction of the intersection of X and Y.

Let $loceps = loc_T(\varepsilon)$. The expression ($enumY\ loceps\ w'$) enumerates the list inY in time $\mathcal{O}(|\mathcal{A}_p| \cdot |u|)$, where u is the longest prefix of w' that is a subword of p. Recall that $|u| \geq \max(Y)$. At first, $enumY$ calls the function $scanprefix'$ to compute $loc = loc_T(u)$ and the list $vnodes$ of nodes visited while traversing T from the $root$ to loc. Then the ordered list $listY = [r \mid r \in Y]$ is computed. For each node $\overline{v}$ in $vnodes$, $|v|$ occurs in $listY$ if and only if $issuffix(\overline{v})$ is true. If u is a suffix of p, then ($locissuffix\ loc$) evaluates to true and the last element of the list $listY$ is $|u| = |loc|$. Finally, $enumY$ transforms $listY$ into inY using the function $num2bool$.

```
enumY::(location α (num,β,bool))→(string α)→[bool]
enumY loceps w' = num2bool 0 listY
                  where (vnodes,loc,s) = scanprefix' loceps w'
                        listY = [d | N es link (d,isp,True)←vnodes] ++
                                [locdepth loc | locissuffix loc]


num2bool::num→[num]→[bool]
num2bool r (d:ds) = True:num2bool (r+1) ds,      if r=d
                  = False:num2bool (r+1) (d:ds), otherwise
num2bool r [] = []
```

The list inX is computed by the function $enumX$ in $\mathcal{O}(|\mathcal{A}_p|)$ time per element on the average. $enumX$ calls a function $enumX'$ with two arguments $loc = loc_T(z)$ and s. zs is a suffix of w and z is the longest prefix of zs that is a subword of p. Let $r = m - |zs|$. $enumX'$ returns the list $[inX_r, \ldots, inX_{m-1}]$. If $s \neq \varepsilon$, then zs is not a subword and hence not a prefix of p. Therefore, $r \notin X$, that is, inX_r is false. If $s = \varepsilon$, then z is a subword of p. In this case, $enumX'$ calls the function $(suffixlocs\ (const\ [\,]))$ to obtain the list $slocs = [loc_r, \ldots, loc_{m-1}]$ of locations of the non-empty suffixes of z. Obviously, $inX_q = locisprefix(loc_q)$ for each $q, r \leq q \leq m-1$. Hence, $(map\ locisprefix\ slocs)$ returns the correct result.

```
enumX::(location α (β,bool,γ))→(string α)→[bool]
enumX loceps w = enumX' (scanprefix loceps w)

enumX'::(location α (β,bool,γ),string α)→[bool]
enumX' (loc,s) = map locisprefix (suffixlocs (const []) loc), if s = []
               = False:enumX' (scanprefix loc (drop 1 s)),    if rootloc loc
               = False:enumX' (scanprefix (linkloc loc) s),   otherwise
```

Example 4.5.4 Let $p = caacc$ and $t = abcbcbcaaccaaccbcb$. Algorithm ECL splits t into the sequence $(w_0, w_1, w_2, w_3) = (abcbc, bcaac, caacc, bcb)$ of subwords. The following table shows inY and the evaluated part of inX for the different pairs of consecutive subwords (w, w').

$(w, w') =$	$(abcbc, bcaac)$	$(bcaac, caacc)$	$(caacc, bcb)$
$inY =$	$[True]$	$[True,\ True]$	$[True]$
$inX =$	$[False,\ \ldots\ ]$	$[False, True,\ \ldots\ ]$	$[True,\ \ldots\ ]$

Hence, there are two occurrences of p in t. One ends at position $(1+1) \cdot m + 1 = 11$ and one ends at position $(2+1) \cdot m + 0 = 15$. □

Let $q = (i+1) \cdot m$. The expression $(intersect\ loceps\ q\ w\ w')$ returns the list of positions j such that $j = q + r$ for some $r \in X \cap Y$. At first the lists inX and inY are computed using the functions $enumX$ and $enumY$ considerably. Then the three lists $[q, q+1 \ldots]$, inY, and inX are zipped. The resulting list contains a triple $(q + r, True, True)$ if and only if $r \in X \cap Y$. Note that in the implementation of $enumX$ we did not have to take care about the fact that only the first $\max(Y) + 1$ elements of the list inX are evaluated. It is simply the laziness that guarantees this (see also Example 4.5.4). The running time of $intersect$ is $\mathcal{O}(|\mathcal{A}_p| \cdot |u|)$ where u is the longest prefix of w' that is a subword of p.

```
intersect::location α (num,bool,bool)→num→(string α)→(string α)→[num]
intersect loceps q w w'
  = [j | (j,True,True)←zip3 [q..] (enumY loceps w') (enumX loceps w)]
```

The function $enumwpairs$ enumerates the list of pairs $((i+1) \cdot m, (w_i, w_{i+1}))$, $0 \leq i \leq k-1$. The main work is done by a function $divide$ which is called with some arguments $w = w_i$ and $s = w_{i+1} \ldots w_k$ to return the list $[w_i, \ldots, w_k]$. Note that if $s = \varepsilon$ and $|w_i| = m$, then $i = k-1$ and $w_k = \varepsilon$ is the last element in this list (cf. Algorithm ECL). The running time of $enumwpairs$ is $\mathcal{O}(n)$. Due to the laziness, there are only two consecutive words w_i and w_{i+1} to be stored at the same time. Hence, the space consumption is $\mathcal{O}(m)$.

```
enumwpairs::num→(string α)→[(num,(string α,string α))]
enumwpairs m t = zip2 [m,2 * m..] (zip2 wlist (drop 1 wlist))
                 where wlist = divide (take m t) (drop m t)
                       divide w [] = [w,[]], if m = #w
                                   = [w],    otherwise
                       divide w s = w:divide (take m s) (drop m s)
```

The function *exsCL* implements Chang and Lawler's exact string searching algorithm.

```
exsCL::α→(string α)→(string α)→[num]
exsCL sentinel p t
  = [j | (q,(w,w'))←enumwpairs m t; j←intersect loceps q w w']
    where m = #p
          loceps = LocN (cst (p++[sentinel],m+1) (depthisprefixissuffix m))
```

Initially, the compact suffix tree of $p\$$ with the three annotations is computed. For each pair $(q,(w,w'))$ returned by $(enumwpairs\ m\ t)$ the function *intersect* is called to obtain the solutions to the exact string searching problem. The overall running time of *exsCL* is $\mathcal{O}(|\mathcal{A}_p|\cdot m+n+|\mathcal{A}_p|\cdot l\cdot(n/m))$, where l is as in section 4.5. The space requirement is $\mathcal{O}(m)$. Hence, our implementation is optimal.

For Algorithm ECL$'$ we implement a variation of the function *intersect*.

```
intersect'::num→(location α (num,β,bool))→
                (location α (num,β,bool))→num→(string α)→(string α)→[num]
intersect' m loceps loceps' q w w'
  = [j | (j,True,True)←zip3 [q+minx..] inX (drop minx (enumY loceps w'))]
    where inX = (reverse.drop 1.enumY loceps'.reverse) w
          minx = m-#inX
```

intersect$'$ is supplied with an additional argument $loceps' = loc_{T'}(\varepsilon)$ where $T' = cst(p^{-1}\$)$. Let $minx = \min(X)$. The set X is represented by the list $inX = [inX_{minx},\ldots,inX_{m-1}]$ such that inX_r is true if and only if $r \in X$. Note that $minx = m - |inX|$. inX is computed in four phases. At first w is reversed. Then the list $[True, inX_{m-1}, inX_{m-2},\ldots,inX_{minx}]$ is enumerated, using the function $enumY$ and the tree T'. The first element of this list is dropped and the remaining list is reversed. This gives list inX. Note that $r \in X \cap Y$ implies $r \geq \min(X)$. Thus, to determine the intersection of X and Y, it suffices to compute the elements of Y that are $\geq \min(X)$. This is accomplished by dropping the first $minx$ elements of the list inY which is constructed by applying $enumY$ to $loceps'$ and w'.

The function *exsCL*$'$ is an optimal implementation of algorithm ECL$'$.

```
exsCL'::α→(string α)→(string α)→[num]
exsCL' sentinel p t
  = [j | (q,(w,w'))←enumwpairs m t; j←intersect' m loceps loceps' q w w']
    where m = #p
          p' = reverse p
          loceps = LocN (cst (p++[sentinel],m+1) (depthisprefixissuffix m))
          loceps' = LocN (cst (p'++[sentinel],m+1) (depthisprefixissuffix m))
```

4.6 Overview of the Implementations

In this chapter, we presented the most important algorithms for solving the exact string searching problem. We implemented most of the algorithms. Figure 4.5 gives an overview of the programs. All functional implementations are optimal. That is, they achieve the same asymptotic efficiency as their imperative counterparts. Input strings are scanned online from left to right. At each time the access to t is restricted to a small sliding window of size $\mathcal{O}(m)$. A remarkable virtue of our functional implementations is that, except for *exsBM*, *exsBMH*, and *exsBMS*, an explicit buffering mechanism is not required to achieve the space efficiency of $\mathcal{O}(m)$. Lazy evaluation and the memory management of the Miranda system guarantee that only the needed portion of the input string is stored at any time.

Figure 4.5: Overview of the exs-Programs

function	space	running time worst	running time expected	comment
exsBF	$\mathcal{O}(m)$	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m+n)$	generates suffixes; uses built-in predicate "=" to compare pattern and m-gram of t
exsKMP	$\mathcal{O}(m)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$	traverse suffix links of KMP-tree in order to avoid repeating character comparisons; no buffering of t required
exsKR	$\mathcal{O}(m+\lvert\mathcal{A}\rvert)$	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m+n)$	input alphabet must be known in advance; uses hash techniques to avoid character comparisons
exsBM	$\mathcal{O}(m+\lvert\mathcal{A}\rvert)$	$\mathcal{O}(m \cdot \lvert\mathcal{A}_p\rvert + \lvert\mathcal{A}\rvert + m \cdot n)$	$\mathcal{O}(m \cdot \lvert\mathcal{A}_p\rvert + \lvert\mathcal{A}\rvert + n)$	input alphabet must be known in advance; requires extension 1; applies good-suffix and bad-character heuristic; explicit buffering of t
exsBMH	$\mathcal{O}(m+\lvert\mathcal{A}\rvert)$	$\mathcal{O}(m + \lvert\mathcal{A}\rvert + m \cdot n)$	$\mathcal{O}(m + \lvert\mathcal{A}\rvert + n)$	input alphabet must be known in advance; requires extension 2; applies Horspool's heuristic; uses built-in predicate "=" to compare pattern and m-gram of t; explicit buffering of t
exsBMS	$\mathcal{O}(m+\lvert\mathcal{A}\rvert)$	$\mathcal{O}(m + \lvert\mathcal{A}\rvert + m \cdot n)$	$\mathcal{O}(m + \lvert\mathcal{A}\rvert + n)$	input alphabet must be known in advance; requires extension 2; applies Sunday's heuristic; uses built-in predicate "=" to compare pattern and m-gram of t; explicit buffering of t
exsCL	$\mathcal{O}(m)$	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot m + n + \lvert\mathcal{A}_p\rvert \cdot l \cdot (n/m))$		divide t into non-overlapping subwords of length m; traverse $cst(p\$)$ in two different ways; combine features of KMP and BM Algorithm
		$l = m$	$l = \log_{\lvert\mathcal{A}\rvert} m$	
exsCL'	$\mathcal{O}(m)$	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot m + n + \lvert\mathcal{A}_p\rvert \cdot l \cdot (n/m))$		divide t into non-overlapping subwords of length m; traverse $cst(p\$)$ and $cst(p^{-1}\$)$ in the same way; combine features of KMP and BM Algorithm
		$l = m$	$l = \log_{\lvert\mathcal{A}\rvert} m$	

Chapter 5

Multiple Exact String Searching

The multiple exact string searching problem is a generalization of the exact string searching problem. The goal is to look for occurrences of several patterns in a single input string. This problem occurs in different contexts, for instance, in searching bibliographic databases [AC75], in security applications to detect suspicious keywords [WM94], and in the analysis of biosequences [Bos88, Smi88, Sta88, CG89, Ste91].

Definition 5.0.1 Suppose the following items are given:

- A finite set $\mathcal{P} \subseteq \mathcal{A}^*$.
- A string $t \in \mathcal{A}^*$ of length n.

$\mathcal{A}$ is the *input alphabet*, $\mathcal{P}$ is the *pattern set*, and t the *input string*. The *multiple exact string searching problem* is to enumerate all positions in t where some $p \in \mathcal{P}$ ends. These positions are referred to as *solutions* to the multiple exact string searching problem. □

The multiple exact string searching problem is sometimes called dictionary matching problem (see [Bre94]). It has recently gained interest after the development of algorithms that can efficiently handle dynamically changing pattern sets (cf. [AF91, AFM92, IS92]).

If we allow preprocessing of the input string, then we can construct the compact suffix tree of $t\$$ and search each $p \in \mathcal{P}$ separately in $\mathcal{O}(|\mathcal{A}| \cdot |p|)$ time. This leads to an algorithm which will be called suffix tree search algorithm. If we do not allow to preprocess t, then a naive method would be to look for each pattern $p \in \mathcal{P}$ separately, using one of the algorithms described in the previous chapter. However, the disadvantage is that t is scanned $|\mathcal{P}|$ times which leads to a search time of a least $\mathcal{O}(|\mathcal{P}| \cdot n)$. The Aho-Corasick Algorithm [AC75] overcomes this problem by looking for all patterns simultaneously. In the following two sections we present the suffix tree search algorithm and the Aho-Corasick Algorithm in detail. Section 5.3 gives a short description of other approaches to the multiple exact string searching problem.

5.1 The Suffix Tree Search Algorithm

The suffix tree search algorithm is of special interest if the pattern set is large. One hopes that the effort for constructing the compact suffix tree of the input string is amortized by the speed up of the search. If we have constructed $T = cst(t\$)$, then the multiple exact string searching problem can be solved as follows: For each $p \in \mathcal{P}$ evaluate $(loc, s) = scanprefix(loc_T(\varepsilon), p)$. According to Definition 3.2.9, p occurs in T if and only if $s = \varepsilon$. Suppose $s = \varepsilon$. Then $loc = loc_T(p)$. Let $\overline{x} = ceiling(loc)$ and $k = |loc| - |x|$. Obviously, p ends at position j in t if and only if $j = n + 1 - k - |y|$ for some string y such that $\overline{xy}$ is a leaf in T. Hence, it suffices to enumerate the lengths of all paths from $\overline{x}$ to some leaf in T. This can be accomplished in $\mathcal{O}(|T_{\overline{x}}|)$ steps by a single traversal of the subtree $T_{\overline{x}}$. Let occ_p be the number of occurrences of p in t. Obviously, $occ_p \in \mathcal{O}(|T_{\overline{x}}|)$. Hence, the positions in t where p ends can be computed in $\mathcal{O}(|\mathcal{A}| \cdot |p| + occ_p)$ time. Altogether, the suffix tree search algorithm requires $\mathcal{O}(n)$ space and it achieves a running time of $\mathcal{O}(|\mathcal{A}| \cdot n + \sum_{p \in \mathcal{P}}(|\mathcal{A}| \cdot |p| + occ_p))$ where the first summand is for the construction of T.

5.1.1 Implementation

Suppose the node $\overline{x}$ in T is represented by the expression $(N\ es\ link\ tag)$. The function *paths* returns the list $plens = [|y| \mid \overline{xy}$ is a leaf in $T_{\overline{x}}]$ of lengths of the paths from $\overline{x}$ to some leaf in $T_{\overline{x}}$. If $\overline{x}$ is a leaf, then there is only the empty path from $\overline{x}$ to a leaf and $plens = [0]$. Otherwise, for each edge $\overline{x} \xrightarrow{w} node$ and each i returned by $(paths\ node)$ there is an element $l + i$ in *plens* where $l = |w|$. The running time of *paths* is proportional to $|plens|$.

```
paths::(ctree α β)→[num]
paths (N es link tag) = [0],                                    if es = []
                      = [l+i | ((w,l),node)←es; i←paths node], otherwise
```

Suppose $p \in words(T)$ and $loc = loc_T(p)$. *loc2endpos* determines from *loc* the list of positions in t where p ends. This is accomplished as described in the previous section. The running time of *loc2endpos* is proportional to the length of the list returned by the function *paths*.

```
loc2endpos::num→(location α β)→[num]
loc2endpos n (LocN node) = [n+1-i | i←paths node]
loc2endpos n (LocE node v (w,k) node') = [n+1-k-i | i←paths node']
```

exmsSTS implements the suffix tree search algorithm. At first it constructs the compact suffix tree of $t\$$ using the function *cst*. Since we do not need to annotate the tree, the second argument of *cst* is undefined. $\mathcal{P}$ is given as a list *patterns*. For each element in *patterns* the function *scanprefix* is called. If it returns a pair of the form $(loc, [\,])$, then p occurs in T and the positions in t where p ends are computed by evaluating $(loc2endpos\ n\ loc)$.

```
exmsSTS::α→[string α]→(string α)→[[num]]
exmsSTS sentinel patterns t
  = [loc2endpos n loc | p←patterns; (loc,[])←[scanprefix loceps p]]
    where n = #t
          loceps = LocN (cst (t++[sentinel],n+1) undef)
```

Recall that the function *cst* implements the lazy suffix tree construction. Thus, the construction phase is interleaved with the search phase of the algorithm accomplished by the functions *scanprefix* and *loc2endpos*. Paths of T are constructed only when being traversed for the first time. This is a very important feature of our implementation, since the practical running time is not dominated by the construction of the suffix tree, but by the searches.

5.2 The Aho-Corasick Algorithm

The Aho-Corasick Algorithm is a generalization of the Knuth-Morris-Pratt Algorithm. It processes the input string online character by character and looks for all patterns simultaneously. This is accomplished by representing the pattern set by an atomic $\mathcal{A}^+$-tree. Like the Knuth-Morris-Pratt Algorithm, the Aho-Corasick Algorithm traverses at most $2 \cdot n$ edges. If a node $\overline{v}$ is reached such that p is a suffix of v for some $p \in \mathcal{P}$, then a match is found.

Definition 5.2.1 We write $w \sqsubset \mathcal{P}$ if there is a $p \in \mathcal{P}$ such that $w \sqsubset p$. The AC-tree for $\mathcal{P}$ is the atomic $\mathcal{A}^+$-tree T with suffix links and a boolean annotation *accept* such that

- $words(T) = \{w \in \mathcal{A}^* \mid w \sqsubset \mathcal{P}\}$.
- For each node $\overline{v}$ in T, $accept(\overline{v})$ is true iff there is a $p \in \mathcal{P}$ such that p is a suffix of v.

A node $\overline{v}$ is accepting if $accept(\overline{v})$ is true. □

Note that the AC-tree for $\mathcal{P} = \{p\}$ is identical to the KMP-tree for p (see section 4.2.1). An AC-tree T corresponds to a pattern matching machine as defined in [AC75]. However, in the pattern matching machine of Aho and Corasick each node $\overline{v}$ is annotated by the set $\{p \in \mathcal{P} \mid v \int p\}$. This allows to identify the patterns ending at position j in t whenever j is a solution to the multiple exact string searching problem.

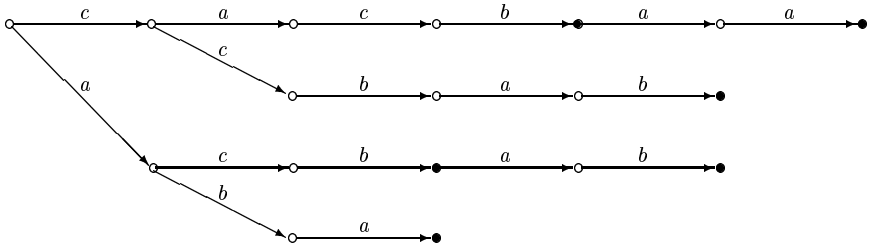
Algorithm AC [AC75] Compute the AC-tree for $\mathcal{P}$. Let $\overline{s_0} = root$. For each $j, 0 \leq j \leq n-1$, compute $\overline{s_{j+1}} = next(\overline{s_j}, t_{j+1})$, where the function *next* is defined as in section 3.7.1. If $accept(\overline{s_j})$ is true for some $j, 0 \leq j \leq n$, then output j. □

The AC-tree together with the function *next* can be considered as a space efficient representation of a DFA. The nodes of are the states and *next* is the state transition function.

Theorem 5.2.2 Algorithm AC correctly solves the multiple exact string searching problem.

Proof We first show by induction on j that s_j is the longest string such that $t_1 \ldots t_j \int s_j$ and $s_j \sqsubset \mathcal{P}$ (see [AC75, Lemma 3]). For $j = 0$ the claim immediately follows. Suppose the claim holds for some $j, 0 \leq j \leq n-1$. Let $\overline{s_{j+1}} = next(\overline{s_j}, t_{j+1})$. By Lemma 3.7.2 s_{j+1} is the longest suffix of $s_j t_{j+1}$ such that $\overline{s_{j+1}}$ is a node in T. Hence, by Definition 5.2.1, s_{j+1} is the longest string such that $s_j t_{j+1} \int s_{j+1}$ and $s_{j+1} \sqsubset \mathcal{P}$. Note that $s_j t_{j+1}$ is a suffix of $t_1 \ldots t_{j+1}$, since s_j is a suffix of $t_1 \ldots t_j$ by induction hypothesis. Hence, s_{j+1} is also a suffix of $t_1 \ldots t_{j+1}$. This proves the induction step. Now suppose that j is a solution to the multiple exact string searching problem. Then there is some $p \in \mathcal{P}$ such that p is a suffix of $t_1 \ldots t_j$. From the result above we can conclude that p is a suffix of s_j. Hence, $accept(\overline{s_j})$ is true and Algorithm AC outputs j. □

Figure 5.1: The AC-Tree for $\{cacbaa, acb, aba, acbab, ccbab\}$.



Example 5.2.3 Let $\mathcal{P} = \{cacbaa, acb, aba, acbab, ccbab\}$. The corresponding AC-tree is shown in Figure 5.1. The accepting nodes are shown as thick circles. □

Example 5.2.3 is taken from [Aho90]. However, we emphasize that there is a small but important difference. Since $acb \in \mathcal{P}$, the node $\overline{cacb}$ is an accepting node, according to Definition 5.2.1. This contrasts [Aho90, page 275] where an accepting node is defined as follows: each node corresponding to a pattern is an accepting node. Consequently, the node $\overline{cacb}$ would not be accepting, since $cacb \notin \mathcal{P}$ (see [Aho90, Example 4.4]). This would mean that for $t = cacb$ Algorithm AC would not output position 4, which is incorrect, since the pattern acb ends at this position.

The AC-tree can be constructed by inserting the patterns, one after the other, into an initially empty tree. This takes $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot r)$, where $r = \sum_{p \in \mathcal{P}} |p|$ and $\mathcal{A}_\mathcal{P}$ is the set of characters occurring in the patterns. Alternatively, one can modify the lazy suffix tree algorithm such that it constructs an atomic $\mathcal{A}^+$-tree. This leads to a construction time of $\mathcal{O}(r^2)$.

The suffix links of the AC-tree are obtained in $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot r)$ steps, as described in section 3.7.1. The annotation *accept* can be computed in constant time per node as follows. Let $\overline{v}$ be a node in T. If $\overline{v}$ is the *root*, then $accept(\overline{v})$ is true if and only if $v \in \mathcal{P}$. Otherwise, $accept(\overline{v})$ is true if and only if $v \in \mathcal{P}$ or $accept(\overline{w})$ is true, where $\overline{v} \longrightarrow \overline{w}$ is the suffix link for $\overline{v}$ in T. To obtain the annotation in this way $accept(\overline{w})$ must be determined before $accept(\overline{v})$. This can be accomplished by computing the annotation in a breadth first traversal of T. Altogether, the construction of T including suffix links and annotation can be done in $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot r)$ time.

The running time of the search phase is determined by the number of calls to the function *next*. In each call of the form $next(\overline{s}, t_{j+1})$ either (1) the input character t_{j+1} is consumed, or (2) the suffix link $\overline{s} \longrightarrow \overline{v}$ is traversed. The number of steps of type (1) is bound by n. Moreover, for each step of type (2) there are $|s| - |v| > 0$ steps of type (1). Thus, the number of steps of type (2) is bound by n, too. Therefore, there are at most $2 \cdot n$ calls to the function *next*. Each call requires $\mathcal{O}(|\mathcal{A}_\mathcal{P}|)$ time in the worst case. Hence, the overall running time of Algorithm AC is $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot (n + r))$. The space requirement is $\mathcal{O}(r)$.

5.2.1 Implementation

The function *treeinsert* takes a string v and inserts it into an atomic $\mathcal{A}^+$-tree. The resulting tree contains a node $\overline{v}$ which is labeled by the value *True*. The running time of *treeinsert* is $\mathcal{O}(l \cdot |v|)$ where l is the average number of edges outgoing from the visited nodes.

```
treeinsert::(tree α bool)→(string α)→tree α bool
treeinsert (N es link tag) [] = N es link True
treeinsert (N es link tag) (a:w)
  = N (edgeinsert es) link tag
    where edgeinsert ((c,node):es) = (c,node):edgeinsert es,      if a > c
                                   = (a,nobranch w):(c,node):es,  if a < c
                                   = (c,treeinsert node w):es,    otherwise
          edgeinsert [] = [(a,nobranch w)]
```

We suppose the pattern set $\mathcal{P}$ is given as a list *patterns*. *patterns2tree* iteratively applies the function *treeinsert* to all patterns. This yields an atomic $\mathcal{A}^+$-tree T' such that $words(T') = \{w \in \mathcal{A}^* \mid w \sqsubset \mathcal{P}\}$. The running time of *patterns2tree* is $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot r)$, where $r = \sum_{p \in \mathcal{P}} |p|$.

```
patterns2tree::[string α]→tree α bool
patterns2tree patterns = foldl treeinsert (N [] undef False) patterns
```

Note that the tree T' returned by *patterns2tree* has undefined suffix links. Moreover, each node $\overline{v}$ is labeled with the value *True* if and only if $v \in \mathcal{P}$. To set the suffix links in T', we could use the function (*addlinks setlink*) as defined in section 3.7.1. In a second phase we could compute the annotation *accept*, since then the suffix links are available. However, a little thought shows that in the second phase we would also have to update the suffix links, since changing some node labels means that previous suffix links may point to obsolete subtrees. For this reason, we merge the setting of the suffix links and the computation of the annotation. This leads to a function *setlink'*.

```
setlink'::(tree α bool)→(tree α bool)→tree α bool
setlink' (N es link tag) link'
  = N es' link' (tag ∨ seltag link')
    where es' = [(a,setlink' node (next link' a)) | (a,node)←es]
```

The function *setlink'* is similar to the function *setlink* (see section 3.7.1). *setlink'* additionally computes the annotation *accept* as described in section 5.2. Thus, if we apply (*addlinks setlink'*) to T', then we obtain the AC-tree T for $\mathcal{P}$. Let us briefly explain how *setlink'* works: Suppose the node $\overline{v}$ in T' is represented by the expression (*N es link tag*). Moreover, let $\overline{v} \longrightarrow \overline{w}$ be the suffix link for $\overline{v}$ and assume that $\overline{w}$ is represented in T by the expression *link'*. Then (*seltag link'* = *accept*($\overline{w}$)) and *accept*($\overline{v}$) is true if and only if *tag* or (*seltag link'*) evaluates to true. From Theorem 3.7.3 we conclude that the overall running time of (*addlinks setlink'*) is $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot r)$. Due to the laziness, we do not have to worry about the order in which the suffix links and the annotations are computed. Note that an implementation in an eager language (cf. [AC75, Algorithm 3]) requires added complication, since the computation must be arranged in an order that is compatible with the dependencies of the suffix links and the annotations (see the remark on page 63).

exmsAC is an optimal implementation of Algorithm AC. In a first phase *exmsAC* constructs the tree structure. In a second phase the suffix links and the annotation *accept* is calculated. This yields the AC-tree for $\mathcal{P}$ which is traversed using the function *iternext*. Recall that *iternext* was already used for implementing the Knuth-Morris-Pratt Algorithm (see section 4.2.1). This emphasizes the strong resemblance of both algorithms.

```
exmsAC::[string α]→(string α)→[num]
exmsAC = iternext.addlinks setlink'.patterns2tree
```

5.3 Other Approaches

The basic idea of the Commentz-Walter Algorithm [CW79] is to construct an atomic $\mathcal{A}^+$-tree T for the reversed patterns. T is used to keep track of the suffixes of the patterns matching the current part of the input string. In particular, to check if there is a match ending at position j, T is traversed, directed by the characters t_j, t_{j-1}, and so on, until a node in T is reached which does not have a suitable outgoing edge. A shift value, by which T can be moved to the right, without loosing a solution, is computed similarly to the Boyer-Moore Algorithm. The preprocessing time of the Commentz-Walter Algorithm is $\mathcal{O}(|\mathcal{A}_\mathcal{P}| \cdot r)$, where $r = \sum_{p \in \mathcal{P}} |p|$. The search takes $\mathcal{O}(n \cdot r)$ steps in the worst case. If $\mathcal{P}$ is small, then the algorithm is expected to be faster then the Aho-Corasick Algorithm. Baeza-Yates and Régnier [BYR90] show that the Commentz-Walter Algorithm can be simplified and speeded up by computing the shift values according to Horspool's heuristic.

The method of Kim and Shawe-Taylor [KST92b] is based on q-grams. The idea is to build an atomic $\mathcal{A}^+$-tree T that contains the reversed q-grams of the patterns together with corresponding shift information. To determine if j is a solution to the multiple exact string searching problem, T is traversed, directed by the reverse of the q-gram $t_{j-q+1} \ldots t_j$. This yields a shift value which aligns $t_{j-q+1} \ldots t_j$ with its rightmost occurrence in some $p \in \mathcal{P}$ (if such an occurrence exists), or it aligns a suffix of $t_{j-q+1} \ldots t_j$ with its occurrence as a prefix of some $p \in \mathcal{P}$. This strategy is very similar to the good-suffix heuristic of the Boyer-Moore Algorithm. Kim and Shawe-Taylor give a detailed complexity analysis for their algorithm. Under reasonable assumptions, they show that the expected running time is $\mathcal{O}((n/m + |\mathcal{P}| \cdot m) \cdot \log(|\mathcal{P}| \cdot m))$.

Wu and Manber [WM94] devised an algorithm that generalizes Horspool's heuristic to q-grams. This allows to handle large pattern sets efficiently. Let $m = \min\{|p| \mid p \in \mathcal{P}\}$ and q be a positive integer in the order of $\log_{|\mathcal{A}|}(2 \cdot r)$. Wu and Manber recommend to choose $q = 2$ if $\mathcal{P}$ is small and $q = 3$ if $\mathcal{P}$ is large. The central feature of the algorithm is a function $qmove$. For each $w \in \mathcal{A}^q$, $qmove(w)$ is defined as follows: if w is a subword of some $p \in \mathcal{P}$, then $qmove(w) = m - \max\{i \mid \exists p \in \mathcal{P} : w \text{ ends in } p \text{ at position } i\}$. Otherwise, $qmove(w) = m - q + 1$. To save character comparisons, each q-gram is hash coded. In order to check if j is a solution to the multiple exact string searching problem, $k = qmove(t_{j-q+1} \ldots t_j)$ is evaluated. If $k > 0$, then $t_{j-q+1} \ldots t_j$ does not occur as a suffix of any $p \in \mathcal{P}$. Hence, j is not a solution to the multiple exact string searching problem and the algorithm moves k positions to the right. In typical situations this case occurs most of the time. If $k \leq 0$, then $t_{j-q+1} \ldots t_j$ is a suffix of some $p \in \mathcal{P}$. To verify, if j is indeed a solution to the multiple exact string searching problem, the algorithm uses two additional precomputed functions, to determine a small subset of $\mathcal{P}$, containing all candidates which may end at position j. These candidates are directly compared to the corresponding suffix of $t_1 \ldots t_j$. If there is a match, it is reported and the algorithm moves one position to the right. The worst case running time of the algorithm is $\mathcal{O}(n \cdot r)$. However, in the expected case only $\mathcal{O}((q \cdot n)/m)$ steps are needed. Wu and Manber implemented their algorithm as part of the *glimpse* system [MW93]. They found it to be very fast in practice, even for thousands of patterns.

Chapter 6

Approximate String Searching

In this chapter, we consider the problem of finding approximate occurrences of a pattern in an input string. This problem occurs in many different contexts. For instance, if one looks up a name in a directory, one may not remember the exact spelling of a name, or the name may be misspelled. Another example occurs in biological sequence analysis. A genomic database is likely to contain DNA or protein sequences of many individuals. Therefore, matching a piece of DNA against the database must allow a small but significant error due to the differences in DNA among individuals of the same or related species. Furthermore, current DNA sequencing techniques are not perfect, and experimental error can sometimes contribute as much as 5-10 percent inaccuracies.

There are several distance models to measure the approximation quality. The most common model is the edit distance model. Some authors also use the hamming distance model which counts the number of mismatches of two strings of the same length. This leads to the k-mismatches problem which is, for instance, studied in [LV86, GG88, OM88, KST92a, TU93, BYP92, PW95]. Note that the hamming distance can be viewed as an edit distance with a cost function that charges replacements with low costs and indels with high costs. Ukkonen [Ukk92a] studies approximate string searching in the q-gram model and the maximal matches model. However, the main motivation for this was to speed up approximate string searching algorithms that are based on the edit distance. Recently, Zobel and Dart [ZD95] have practically evaluated the different distance models for finding approximate matches in large lexicons. In this chapter, we measure the approximation quality by the edit distance. This leads to the following problem definition.

Definition 6.0.1 Suppose the following items are given:

- A cost function δ.
- A non-negative real value k.
- A string $p \in \mathcal{A}^*$ of length m.
- A string $t \in \mathcal{A}^*$ of length n.

$\mathcal{A}$ is the *input alphabet*, k is the *threshold value*, p is the *pattern*, and t is the *input string*. An *approximate match* is a subword v of t such that $edist_\delta(p, v) \leq k$. The *approximate string*

searching problem is to enumerate all positions in t where an approximate match ends. These positions are referred to as *solutions* to the approximate string searching problem. □

As in chapter 4, $\mathcal{A}_p$ denotes the set of characters in p.

In sections 6.1 and 6.2, we describe in detail several algorithms for solving the approximate string searching problem for arbitrary cost functions. In sections 6.3 to 6.8 the k-differences problem is considered. This is the approximate string searching problem restricted to the unit cost function. Except for one, all described algorithms process the input string online from left to right. Two algorithms buffer parts of the input string, but at each time the access to t is restricted to one input character, or to a small sliding window of size $\mathcal{O}(m)$ and $\mathcal{O}(k \cdot m)$, respectively. Our functional implementations preserve these properties of the algorithms. We emphasize that an explicit buffering mechanism is not required to achieve space efficient algorithms. Lazy evaluation and the memory management of the Miranda system guarantee that only the needed portion of the input string is stored at any time. Note that there are simplified forms of the approximate string searching problem which, for instance, ask if there is an approximate occurrence of p in t. Our lazy functional programs can be used to solve these problems without changing their code.

6.1 Sellers' Algorithm

By a slight modification of the Wagner-Fischer Algorithm, Sellers [Sel80] obtained a simple method (SEL for short) to solve the approximate string searching problem. The idea of Sellers was to not charge the insertion of input characters into the empty string. Technically this means to set the entries in the first row of E_δ to 0. This gives a table in which the entry $E_\delta(i,j)$ is the minimal edit distance of $p_1 \ldots p_i$ with some suffix of $t_1 \ldots t_j$. From the last row of this table one can easily obtain the solutions to the approximate string searching problem.

Sellers' variation is usually described by giving the recurrence for an $(m+1) \times (n+1)$-table, similar as in Theorem 3.9.6. Our approach is slightly different. Similar to the functional implementation of the Wagner-Fischer Algorithm, we specify SEL by an initial column and a function that transforms one column into the next column. This means that the specified columns do not only refer to the global input string t, but to some arbitrary string. In section 6.2 and 6.5, we will make use of this notational convenience.

Definition 6.1.1 C denotes the set of functions $f : \{0, \ldots, m\} \to \mathbb{R}_0^+$. The elements of C are *columns*. We define a function $nextdcol : C \times \mathcal{A} \to C$ as follows. For all $f \in C$ and all $b \in \mathcal{A}$,

$$\begin{aligned} nextdcol(f,b) &= f_b \\ &\quad \text{where } f_b(0) = 0 \\ &\qquad f_b(i+1) = \min \left\{ \begin{array}{l} f_b(i) + \delta(p_{i+1} \to \varepsilon) \\ f(i) + \delta(p_{i+1} \to b) \\ f(i+1) + \delta(\varepsilon \to b) \end{array} \right\} \end{aligned}$$

Moreover, we define a function $dcol : \mathcal{A}^* \to C$ as follows:

$$\begin{array}{lcl} dcol(vb) & = & nextdcol(dcol(v), b) \\ dcol(\varepsilon) & = & f \\ & & \text{where } f(0) = 0 \\ & & \qquad f(i+1) = f(i) + \delta(p_{i+1} \to \varepsilon). \end{array}$$

$dcol(v)$ is the *distance column* of v. $f \in C$ is a distance column if $f = dcol(v)$ for some $v \in \mathcal{A}^*$. □

A scheme which is very similar to Definition 6.1.1 already occurred in [JU91]. Note that $dcol(v)$ and $nextdcol(f, b)$ depend on p. This is not reflected by our notation.

Algorithm SEL [Sel80] Compute $dcol(\varepsilon)$. For each $j, 1 \leq j \leq n$ compute

$$dcol(t_1 \ldots t_j) = nextdcol(dcol(t_1 \ldots t_{j-1}), t_j).$$

If $dcol(t_1 \ldots t_j)(m) \leq k$, then output j. □

Theorem 6.1.2 Algorithm SEL correctly solves the approximate string searching problem.

Proof One easily shows by induction that for each string v and for each $i, 0 \leq i \leq m$, the following equality holds:

$$dcol(v)(i) = \min\{edist_\delta(p_1 \ldots p_i, s) \mid s \text{ is a suffix of } v\}$$

Hence, $dcol(t_1 \ldots t_j)(m) \leq k \iff$ there is a suffix s of $t_1 \ldots t_j$ such that $edist_\delta(p, s) \leq k \iff$ there is an approximate match ending at position j. □

In an implementation of Algorithm SEL, every distance column is represented by an array or a list which takes $\mathcal{O}(m)$ space. $dcol(\varepsilon)$ is computed in $\mathcal{O}(m)$ steps. For each $j, 1 \leq j \leq n$ the distance column $dcol(t_1 \ldots t_j)$ is computed in $\mathcal{O}(m)$ steps as well. This gives an overall time efficiency of $\mathcal{O}(m \cdot n)$. Since in every step only the actual and the previous distance column must be stored, SEL needs $\mathcal{O}(m)$ space.

In the following, we show how to improve the average case behavior of SEL. The idea is to compute the distance column of $t_1 \ldots t_j$ modulo some equivalence.

Definition 6.1.3 Let f and f' be distance columns. f and f' are *equivalent*, denoted by $f \equiv f'$, if for all $i, 0 \leq i \leq m$ we have $f(i) = f'(i)$ whenever $f(i) \leq k$ or $f'(i) \leq k$. □

An equivalence notion for columns was introduced in [Ukk93a]. However, it referred to pairs of distance and length columns (see also Definitions 6.2.7 and 6.2.12). The following lemma shows that the relation $\equiv$ is preserved by *nextdcol*.

Lemma 6.1.4 Let $f \equiv f'$ and $b \in \mathcal{A}$. Then $nextdcol(f, b) \equiv nextdcol(f', b)$.

Proof Let $f_b = nextdcol(f, b)$ and $f'_b = nextdcol(f', b)$. By induction on i, we show that $f_b(i) \leq k$ or $f'_b(i) \leq k$ implies $f_b(i) = f'_b(i)$. For $i = 0$ we have $f_b(i) = 0 = f'_b(i)$. Assume that $f_b(i+1) \leq k$ and consider the following cases:

- If $f_b(i+1) = f_b(i)+\delta(p_{i+1} \to \varepsilon)$, then $f_b(i) < k$ which implies $f_b(i) = f'_b(i)$ by induction hypothesis.
- If $f_b(i+1) = f(i)+\delta(p_{i+1} \to b)$, then $f(i) \leq k$ which implies $f(i) = f'(i)$ by assumption.
- If $f_b(i+1) = f(i+1)+\delta(\varepsilon \to b)$, then $f(i+1) < k$ which implies $f(i+1) = f'(i+1)$ by assumption.

Hence, we obtain

$$f_b(i+1) = \min \left\{ \begin{array}{l} f_b(i) + \delta(p_{i+1} \to \varepsilon) \\ f(i) + \delta(p_{i+1} \to b) \\ f(i+1) + \delta(\varepsilon \to b) \end{array} \right\} = \min \left\{ \begin{array}{l} f'_b(i) + \delta(p_{i+1} \to \varepsilon) \\ f'(i) + \delta(p_{i+1} \to b) \\ f'(i+1) + \delta(\varepsilon \to b) \end{array} \right\} = f'_b(i+1)$$

By an analogous argumentation, one shows $f'_b(i+1) = f_b(i+1)$ whenever $f'_b(i+1) \leq k$. □

Lemma 6.1.4 corresponds to Lemma 4 in [Ukk93a]. However, Ukkonen omitted the proof.

Definition 6.1.5 [Ukk93a] Let f be a distance column. An entry $f(i)$ is *essential* if $f(i) \leq k$. $lei(f) = \max\{i \mid 0 \leq i \leq m, f(i) \leq k\}$ is the *last essential index* of f. Let $l = lei(f)$. $f(l)$ is the *last essential entry* in f. □

Note that $f(m) \leq k$ if and only if the last essential index of f is m. Suppose $l = lei(f)$ and $f_b = nextdcol(f, b)$. By Lemma 6.1.4, the essential entries in f_b do not depend on $f(l+1), f(l+2), \ldots, f(m)$ since these values are larger than k. Hence, it is not necessary to always calculate f_b completely, as done by Sellers' Algorithm. The calculation of f_b can be modified as follows: Compute $f_b(0), f_b(1), \ldots, f_b(l)$ according to Definition 6.1.1. If $l < m$, then compute

$$\begin{array}{rcl} f_b(l+1) & = & \min \left\{ \begin{array}{l} f_b(l) + \delta(p_{l+1} \to \varepsilon), \\ f(l) + \delta(p_{l+1} \to b) \end{array} \right\} \\ f_b(l+2) & = & f_b(l+1) + \delta(p_{l+2} \to \varepsilon) \\ f_b(l+3) & = & f_b(l+2) + \delta(p_{l+3} \to \varepsilon) \\ & \vdots & \end{array}$$

until a value $f_b(h)$ is reached such that either $h = m$ or $f_b(h) > k$ holds. Thus, the computation of f_b is cut off at index h. The last essential index of f_b is the maximal $i, 0 \leq i \leq h$ such that $f_b(i) \leq k$. If $b = t_j$ and $f = dcol(t_1 \ldots t_{j-1})$, then j is a solution to the approximate string searching problem if and only if $lei(f_b) = m$.

This modification leads to a cutoff variation of Sellers' method (SELco for short) which was suggested by Ukkonen [Ukk85b] in the context of the unit edit distance. Chang and Lampe [CL92] showed that Ukkonen's cutoff trick leads to an average case time efficiency of $\mathcal{O}(k \cdot n)$ if one assumes the unit cost function. In [Ukk93a], it is implicitly stated that this efficiency result holds for an arbitrary cost function as well. Note that the cutoff variation does not improve the worst case efficiency of $\mathcal{O}(m \cdot n)$.

6.1.1 Implementation

Since Sellers' method differs from the Wagner-Fischer Algorithm only by the computation of the first column value, we can take the table specification *edisttable* and modify the function *firstentry* appropriately. This gives the following table specification for SEL.

```
seltable::(costfunction α)→(string α)→tablespec α num
seltable delta p
  = (firstcol,nextdcol)
    where firstcol = 0:[no + delta (D a) | (no,a)←zip2 firstcol p]
          firstentry we b = 0
          join3 a b we no nw
            = min2 (min2 (we+delta (I b)) (no+delta (D a))) (nw+delta (R a b))
          nextdcol = absnextcol firstentry join3 undef p
```

Recall that a column is represented by a list. So $dcol(\varepsilon)$ is represented by *firstcol*. The function *appSEL* implements Sellers' method. (The prefix *app* stands for *approximate string searching*.) By evaluating the expression (*scanl nextdcol firstcol t*), the distance columns are enumerated. j is a solution to the approximate string searching problem if and only if $f(m) \leq k$ where $f = dcol(t_1 \ldots t_j)$. The access to $f(m)$, using the predefined list operator (!), takes $\mathcal{O}(m)$ time. However, this is amortized by the computation of f in $\mathcal{O}(m)$ steps.

```
appSEL::(costfunction α)→num→(string α)→(string α)→[num]
appSEL delta k p t
  = [j | (j,f)←zip2 [0..] (scanl nextdcol firstcol t); k >= (f!m)]
    where (firstcol,nextdcol) = seltable delta p
          m = #p
```

The function *selcotable* returns the table specification for the cutoff variation of Sellers' method.

```
selcotable::(costfunction α)→num→(string α)→tablespec α num
selcotable delta k p
  = (firstcol,nextdcol)
    where firstcol = 0:takewhile (<=k) [no+delta (D a) | (no,a)←zip2 firstcol p]
          firstentry we b = 0
          join3 a b we no nw
            = min2 (min2 (we+delta (I b)) (no+delta (D a))) (nw+delta (R a b))
          join2 a b no nw s
            = takewhile (<=k) z
              where z = x:[no+delta (D a) | (no,a)←zip2 z s]
                        where x = min2 (no + delta (D a)) (nw + delta (R a b))
          nextdcol f b = cutsuffix (>k) f'
                         where f' = absnextcol firstentry join3 join2 p f b
```

For the cutoff variation, a distance column f is represented by the list $[f(0), f(1), \ldots, f(l)]$ where $l = lei(f)$. So *firstcol* represents the maximal prefix of $[dcol(\varepsilon)(0), \ldots, dcol(\varepsilon)(m)]$ that consists of values $\leq k$. *firstcol* can easily be computed using the function (*takewhile* $(\leq k)$).

Suppose f is a distance column and $f_b = nextdcol(f, b)$ is to be computed. Recall that $f_b(0), f_b(1), \ldots, f_b(l)$ are calculated according to Definition 6.1.1. Hence, *firstentry* and $join3$ are taken literally from *seltable*. We additionally have defined the function $join2$ which is called whenever $l < m$. $join2$ computes a list z and returns the longest prefix of z that consists of values $\leq k$. The first element of z is $x = f_b(l+1)$ which depends on the value $f_b(l)$ in the "north" and the value $f(l)$ in the "northwest". The remainder of z is the list $[f_b(l+2), f_b(l+2), \ldots]$. If $join2$ returns the empty list and $f_b(l) > k$, then the list f' above has a non-empty suffix which consists of values greater than k. This suffix is cut off in $\mathcal{O}(|f'|)$ steps using the function $(cutsuffix\ (> k))$.

```
cutsuffix::(α→bool)→(column α)→(column α)
cutsuffix bad = foldr front []
                where front value [] = [], if bad value
                      front value f = value:f
```

appSELco implements the cutoff variation of Sellers' method. In order to decide if a column f corresponds to a position where an approximate match ends, it is necessary to test the condition $lei(f) = m$. This is accomplished by dropping the first m elements of f. The remaining list is not empty if and only if $lei(f) = m$.

```
appSELco::(costfunction α)→num→(string α)→(string α)→[num]
appSELco delta k p t
  = [j | (j,f)←zip2 [0..] (scanl nextdcol firstcol t); ok f]
    where (firstcol,nextdcol) = selcotable delta k p
          ok f = ([] ~= drop (#p) f)
```

It is easily verified that both *appSEL* and *appSELco* are optimal implementations of SEL and SELco, respectively.

6.2 Memorizing Distance Columns

In some applications, one wants to solve the approximate string searching problem for a fixed input string but a varying pattern and threshold value.[1] A brute force method for this case is to precompute t into a suffix tree and to make a depth-first traversal over this tree that finds all subwords v of t such that $edist_\delta(p, v) \leq k$. Each path in the tree is followed until the edit distance between the corresponding string and all prefixes of p is larger than k. To determine these edit distances, the corresponding column of table E_δ is computed each time a character in the suffix tree is scanned. The total number of columns evaluated in such a way is $\Theta(m \cdot n)$. Thus, the brute force method does not improve on Sellers' method, in general.

In [Ukk93a], another method is presented which performs a more efficient search. It evaluates at most n distance columns. The key to this method is the observation that the essential entries in $dcol(t_1 \ldots t_j)$ depends only on a relatively short suffix of $t_1 \ldots t_j$, the shortest

[1] Consider, for instance, the intensive study of a biosequence. Different aspects of the analysis often means to search for different approximate patterns with different thresholds.

essential suffix. If this suffix occurs again in t, say as a suffix of $t_1 \dots t_{j'}$ for some $j' > j$, then $dcol(t_1 \dots t_{j'})$ is equivalent to $dcol(t_1 \dots t_j)$. In a lot of cases, Ukkonen's method recognizes this and skips the computation of $dcol(t_1 \dots t_{j'})$, thus improving the search time. Unfortunately, the technique requires to store all computed distance columns, which often leads to a large space consumption.

In the following section, we establish the basis for Ukkonen's method from a declarative point of view. We introduce the notion of essential suffixes and state their properties. Some of these were already given in [Ukk93a]. However, we make explicit some properties implicitly used in [Ukk93a] and provide simpler proofs. Moreover, we describe an algorithm to efficiently compute shortest essential suffixes together with a sequence of distance columns which can be used to solve the approximate string searching problem. Unlike the presentation in [Ukk93a], our presentation of the algorithm does not assume a concrete data structure. This makes it quite amendable for a detailed correctness proof.

6.2.1 Essential Suffixes

The following lemma states that the entries in a distance column are not increasing if one extends a string to the left. Since our distance column notation is independent of the global input string t, this property can conveniently be expressed.

Lemma 6.2.1 Let v be a string and b be a character. Then $dcol(v)(i) \geq dcol(bv)(i)$ for all $i, 0 \leq i \leq m$.

Proof

$$\begin{aligned} dcol(v)(i) &= \min\{edist_\delta(p_1 \dots p_i, s) \mid v \sqsupset s\} \\ &\geq \min(\{edist_\delta(p_1 \dots p_i, s) \mid v \sqsupset s\} \cup \{edist_\delta(p_1 \dots p_i, bv)\}) \\ &= \min\{edist_\delta(p_1 \dots p_i, s) \mid bv \sqsupset s\} \\ &= dcol(bv)(i). \quad \square \end{aligned}$$

From this lemma we can conclude that $dcol(v)(i) \geq dcol(wv)(i)$ for each string w and each $i, 0 \leq i \leq m$. This property will be exploited several times. Note that Ukkonen implicitly used this property, too, but did not state it.

Definition 6.2.2 Let v be a string. A suffix s of v is *essential* if $dcol(s) \equiv dcol(v)$. $ses(v)$ denotes the shortest essential suffix of v. $\square$

Recall that a distance column depends on δ and p. The essential entries in a distance column additionally depends on k. Hence, $ses(v)$ depends on δ, p, and k. Since p and k are arbitrary but fixed, we omit them in our notation. $ses(v)$ determines the essential entries in $dcol(v)$. In other words, if $ses(v) = ses(v')$ then $dcol(v) \equiv dcol(ses(v)) = dcol(ses(v')) \equiv dcol(v')$ (see [Ukk93a, Theorem 1]).

Example 6.2.3 Let δ be the unit cost function and $p = abbb$. Suppose $v = bbaba$. Then we have $dcol(bbaba) = 00112$, $dcol(baba) = 00112$, $dcol(aba) = 00112$, $dcol(ba) = 00123$,

$dcol(a) = 00123$, and $dcol(\varepsilon) = 01234$. If $k = 0$, then *bbaba*, *baba*, *aba*, *ba*, and *a* are the essential suffixes of v. Hence, $ses(v) = a$. If $k > 0$, then *bbaba*, *baba*, and *aba* are the essential suffixes of v. Hence, $ses(v) = aba$. □

In the terminology of [Ukk93a], $ses(t_1 \ldots t_j)$ is the viable (k-approximate) prefix (of p) at j. We have not adopted Ukkonen's terminology since it reflects a property of $ses(t_1 \ldots t_j)$ to be clarified later in our presentation. Note that Ukkonen defines the notion viable prefix in an operational way using length columns (see Definition 6.2.7). Lemma 6.2.8 shows that both notions are equivalent.

The following three lemmas state some basic properties of essential suffixes. Lemma 6.2.4 shows that the last essential entry of a distance column determines whether a suffix is essential or not. Ukkonen exploits this property for the shortest essential suffix. Lemma 6.2.5 corresponds to Theorem 2 in [Ukk93a]. The "$\Leftarrow$"-direction of Lemma 6.2.6 is a slight generalization of Lemma 3 in [Ukk93a].

Lemma 6.2.4 Let v be a string and $h = lei(dcol(v))$. For each suffix s of v, the following statements are equivalent:

(1) s is essential.

(2) $s \int ses(v)$.

(3) $dcol(s)(h) = dcol(v)(h)$.

Proof

(1) $\Rightarrow$ (2) Let s be essential. Then $|s| \geq |ses(v)|$. Since s and $ses(v)$ are suffixes of v, we have $s \int ses(v)$.

(2) $\Rightarrow$ (3) Let $s \int ses(v)$. Then $dcol(ses(v))(h) \geq dcol(s)(h) \geq dcol(v)(h)$ holds. Since $dcol(v)(h) \leq k$, we obtain $dcol(v)(h) = dcol(ses(v))(h)$. Therefore, $dcol(s)(h) = dcol(v)(h)$.

(3) $\Rightarrow$ (1) Let $dcol(s)(h) = dcol(v)(h)$. Suppose s' is the shortest suffix of v such that $dcol(v)(h) = edist_\delta(p_1 \ldots p_h, s')$. Since $edist_\delta(p_1 \ldots p_h, s') = dcol(s)(h)$, s' is a suffix of s. Obviously, s' is the shortest suffix of s such that $dcol(s)(h) = edist_\delta(p_1 \ldots p_h, s')$. Let $0 \leq i \leq m$ and $dcol(v)(i) \leq k$ or $dcol(s)(i) \leq k$. If $dcol(s)(i) \leq k$, then $dcol(v)(i) \leq k$ since $dcol(s)(i) \geq dcol(v)(i)$. Hence, $i \leq h$. Let s'' be the shortest suffix of v such that $dcol(v)(i) = edist_\delta(p_1 \ldots p_i, s'')$. Since $i \leq h$, s'' is a suffix of s'. Thus, s'' is also a suffix of s which implies $dcol(v)(i) = edist_\delta(p_1 \ldots p_i, s'') \geq dcol(s)(i)$. This implies $dcol(s)(i) = dcol(v)(i)$. Hence, $dcol(v) \equiv dcol(s)$, that is, s is essential. □

Lemma 6.2.5 Let v be a string and b a character. Then $ses(vb)$ is a suffix of $ses(v)b$.

Proof Since $v \int ses(v)$, we have $vb \int ses(v)b$. From $dcol(v) \equiv dcol(ses(v))$ and Lemma 6.1.4 we conclude $dcol(vb) = nextdcol(dcol(v), b) \equiv nextdcol(dcol(ses(v)), b) = dcol(ses(v)b)$. Hence, $ses(v)b$ is an essential suffix of vb. Since $ses(vb)$ is the shortest essential suffix of vb, we obtain $ses(v)b \int ses(vb)$. □

Lemma 6.2.6 Let v and v' be strings. $ses(v) = ses(v')$ if and only if v and v' have a common essential suffix.

Proof The "⇒"-direction is trivial. To show the "⇐"-direction, suppose that s is an essential suffix of v and of v'. Then $s \sqsupset ses(v)$ and $s \sqsupset ses(v')$. Let $s' = ses(s)$. Obviously, $dcol(s) \equiv dcol(v)$ and $s' \sqsupset ses(v)$ or $ses(v) \sqsupset s'$. Assume that $s' \neq ses(v)$. Then $dcol(v) \equiv dcol(ses(v)) \not\equiv dcol(s') \equiv dcol(s)$. This is a contradiction. Hence, $s' = ses(v)$. In analogy, one shows $s' = ses(v')$. This implies $ses(v) = ses(v')$. □

To compute shortest essential suffixes, we introduce *length columns*:

Definition 6.2.7 For all strings v, we define the column $lcol(v)$ by

$$lcol(v)(i) = \min\{|s| \mid v \sqsupset s, edist_\delta(p_1 \ldots p_i, s) = dcol(v)(i)\}$$

for each $i, 0 \leq i \leq m$. Thus, $lcol(v)(i)$ is the length of the shortest suffix of v, whose edit distance from $p_1 \ldots p_i$ is $dcol(v)(i)$. $lcol(v)$ is the *length column* of v. $g \in C$ is a length column if $g = lcol(v)$ for some $v \in \mathcal{A}^*$. □

The sequence of length columns for all prefixes of t corresponds to table L in [JU91, Ukk93a].

Lemma 6.2.8 Let $h = lei(dcol(v))$. Then $lcol(v)(h) = |ses(v)|$.

Proof

$$\begin{aligned} lcol(v)(h) &= \min\{|s| \mid v \sqsupset s, edist_\delta(p_1 \ldots p_h, s) = dcol(v)(h)\} \\ &= \min\{|s| \mid v \sqsupset s, dcol(s)(h) = dcol(v)(h)\} \\ &= \min\{|s| \mid v \sqsupset s, dcol(s) \equiv dcol(v)\} \\ &= \min\{|s| \mid v \sqsupset s, s \text{ is essential}\} \\ &= |ses(v)|. \quad \square \end{aligned}$$

From Lemma 6.2.8 we conclude that $edist_\delta(p_1 \ldots p_i, ses(t_1 \ldots t_j)) = dcol(v)(i)$ where $p_1 \ldots p_i$ is the longest prefix of p such that $dcol(t_1 \ldots t_j)(i) \leq k$. For this reason Ukkonen calls $ses(t_1 \ldots t_j)$ the viable k-approximate prefix of p at j.

In analogy to *nextdcol*, we introduce a function for computing length columns.

Definition 6.2.9 For all $f, g \in C$ and all $b \in \mathcal{A}$, the function $nextlcol : C \times C \times \mathcal{A} \to C$ is defined as follows: $nextlcol(f, g, b) = g_b$ where $g_b(0) = 0$ and

$$g_b(i+1) = \begin{cases} g_b(i), & \text{if } f_b(i+1) = f_b(i) + \delta(p_{i+1} \to \varepsilon) \\ g(i)+1, & \text{else if } f_b(i+1) = f(i) + \delta(p_{i+1} \to b) \\ g(i+1)+1, & \text{else if } f_b(i+1) = f(i+1) + \delta(\varepsilon \to b) \end{cases}$$
$$\text{where } f_b = nextdcol(f, b)$$

This recurrence is also given in [JU91, Ukk93a].

Lemma 6.2.10 For each $i, 0 \leq i \leq n$ we have $lcol(\varepsilon)(i) = 0$. For each string v and each character b we have $lcol(vb) = nextlcol(dcol(v), lcol(v), b)$.

Proof Routine. □

Figure 6.1: Distance Columns and Length Columns for $p = abbb$ and $t = a^8b^8$

	dcol															
		a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b	2	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
b	3	2	2	2	2	2	2	2	2	1	0	1	1	1	1	1	1
b	4	3	3	3	3	3	3	3	3	2	1	0	1	1	1	1	1

	lcol															
		a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
b	0	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1
b	0	1	1	1	1	1	1	1	1	2	3	2	2	2	2	2	2
b	0	1	1	1	1	1	1	1	1	2	3	4	3	3	3	3	3

As can easily be verified, a length column can be computed in $\mathcal{O}(m)$ steps. Since length columns depend on the corresponding distance columns, both will be computed simultaneously (see section 6.2.5).

Example 6.2.11 [Ukk93a] Let δ be the unit cost function. Suppose $p = abbb$ and $t = a^8b^8$. The distance columns and the length columns of all prefixes of t are shown in Figure 6.1. Let $k = 1$. Then $ses(a^i) = a$, $1 \leq i \leq 8$, $ses(a^8b) = ab$, $ses(a^8bb) = abb$, $ses(a^8bbb) = abbb$, and $ses(a^8b^i) = bbb$, $4 \leq i \leq 8$. Hence, there are five different shortest essential suffixes. □

The notion of equivalence extends to pairs of columns:

Definition 6.2.12 [Ukk93a] Two pairs (f, g) and (f', g') of columns are *equivalent*, denoted by $(f, g) \equiv (f', g')$, if for each $i, 0 \leq i \leq m$, $f(i) = f'(i)$ and $g(i) = g'(i)$ whenever $f(i) \leq k$ or $f'(i) \leq k$. □

nextlcol preserves equivalence as well, that is, if $(f, g) \equiv (f', g')$ then for all characters b we have $(nextdcol(f, b), nextlcol(f, g, b)) \equiv (nextdcol(f', b), nextlcol(f', g', b))$.

We now present an algorithm that efficiently computes $ses(t_1 \ldots t_j)$ for each $j, 0 \leq j \leq n$. As a by-product, we get for each j a distance column f_j that is equivalent to $dcol(t_1 \ldots t_j)$. This can be used to solve the approximate string searching problem. If $f_j(m) \leq k$, then j is a solution to the approximate string searching problem. The algorithm memorizes, in a lot of cases, that the shortest essential suffix of $t_1 \ldots t_j$ and a distance column equivalent to $dcol(t_1 \ldots t_j)$ was already computed in a previous step. Therefore, we call the algorithm MDC which means *memorizing distance columns*.

Algorithm MDC Set $s_0 = \varepsilon$, $f_0 = dcol(\varepsilon)$, $g_0 = lcol(\varepsilon)$, $V_0 = \{\varepsilon\}$, $D_0 = \{\varepsilon\}$, and $col_0(\varepsilon) = (dcol(\varepsilon), lcol(\varepsilon))$. For each $j, 0 \leq j \leq n-1$ compute $s_{j+1} \in \mathcal{A}^*$, $f_{j+1} \in C$, $g_{j+1} \in C$, $D_{j+1} \subseteq V_{j+1} \subseteq \mathcal{A}^*$, and $col_{j+1} : D_{j+1} \to \mathcal{P}(C \times C)$ as follows:

(*i*) If $s_j t_{j+1} \in V_j$, then s_{j+1} is the longest suffix of $s_j t_{j+1}$ that occurs in D_j. Set $(f_{j+1}, g_{j+1}) = col_j(s_{j+1})$, $V_{j+1} = V_j$, $D_{j+1} = D_j$, and $col_{j+1}(s) = col_j(s)$ for each $s \in D_{j+1}$.

(*ii*) If $s_j t_{j+1} \notin V_j$, then set $(f_{j+1}, g_{j+1}) = (nextdcol(f_j, t_{j+1}), nextlcol(f_j, g_j, t_{j+1}))$. Let $h = lei(f_{j+1})$. s_{j+1} is the suffix of $s_j t_{j+1}$ of length $g_{j+1}(h)$. Moreover, set $V_{j+1} = V_j \cup \{s \mid s_j t_{j+1} \sqsupset s \sqsupset s_{j+1}\}$. If $s_{j+1} \in D_j$, then set $D_{j+1} = D_j$ and $col_{j+1}(s) = col_j(s)$ for each $s \in D_{j+1}$. Otherwise, set $D_{j+1} = D_j \cup \{s_{j+1}\}$ and for each $s \in D_{j+1}$ define

$$col_{j+1}(s) = \begin{cases} (f_{j+1}, g_{j+1}), & \text{if } s = s_{j+1} \\ col_j(s), & \text{otherwise} \end{cases}$$

□

Note that at any time it suffices to store one "generation" of values. More precisely, if the six items s_{j+1}, f_{j+1}, g_{j+1}, V_{j+1}, D_{j+1}, and col_{j+1} are computed, then s_j, f_j, g_j, V_j, D_j, and col_j are no longer needed.

The basic structure of Algorithm MDC is equivalent to Algorithm A in [Ukk93a]. However, Algorithm MDC does not assume a concrete data structure to represent a generation of values. This abstraction simplifies the correctness proof (see below). Moreover, it provides a unified basis for the different implementation techniques considered in the next sections.

Theorem 6.2.13 For each j, $0 \leq j \leq n$ the following holds:

1. $s_j = ses(t_1 \ldots t_j)$
2. $(f_j, g_j) \equiv (dcol(t_1 \ldots t_j), lcol(t_1 \ldots t_j))$
3. $V_j = \{\varepsilon\} \cup \{s \mid ses(v)b \sqsupset s \sqsupset ses(vb)$ for some prefix vb of $t_1 \ldots t_j\}$
4. $D_j = \{ses(v) \mid v$ is a prefix of $t_1 \ldots t_j\}$
5. $col_j(s) \equiv (dcol(s), lcol(s))$ for each $s \in D_j$

Proof By induction on j. For $j = 0$ properties 1-5 obviously hold. Suppose properties 1-5 hold for some $j, 0 \leq j \leq n-1$. Consider the following cases:

(*i*) If $s_j t_{j+1} \in V_j$, then there is a prefix vb of $t_1 \ldots t_j$ such that $ses(v)b \sqsupset s_j t_{j+1} \sqsupset ses(vb)$. By Lemmas 6.2.4 and 6.2.5, $s_j t_{j+1}$ is an essential suffix both of vb and of $t_1 \ldots t_{j+1}$. By Lemma 6.2.6, we get $ses(vb) = ses(t_1 \ldots t_{j+1})$. Moreover, we can conclude that $ses(t_1 \ldots t_{j+1})$ is the longest suffix of $s_j t_{j+1}$ that occurs in D_j. Therefore, $s_{j+1} = ses(t_1 \ldots t_{j+1})$. Furthermore,

$$\begin{aligned} (f_{j+1}, g_{j+1}) &= col_j(s_{j+1}) \\ &\equiv (dcol(s_{j+1}), lcol(s_{j+1})) \\ &\equiv (dcol(t_1 \ldots t_{j+1}), lcol(t_1 \ldots t_{j+1})) \end{aligned}$$

Finally, properties 3, 4, and 5 hold for V_{j+1}, D_{j+1}, and col_{j+1}.

(*ii*) If $s_j t_{j+1} \notin V_j$, then

$$\begin{aligned}(f_{j+1}, g_{j+1}) &= (nextdcol(f_j, t_{j+1}), nextlcol(f_j, g_j, t_{j+1})) \\ &\equiv \begin{array}{l}(nextdcol(dcol(t_1 \ldots t_j), t_{j+1}), \\ \ nextlcol(dcol(t_1 \ldots t_j), lcol(t_1 \ldots t_j), t_{j+1}))\end{array} \\ &\equiv (dcol(t_1 \ldots t_{j+1}), lcol(t_1 \ldots t_{j+1}))\end{aligned}$$

Let $h = lei(f_{j+1})$. Then $h = lei(dcol(t_1 \ldots t_{j+1}))$ and by Lemma 6.2.8 we conclude $g_{j+1}(h) = lcol(t_1 \ldots t_{j+1})(h) = |ses(t_1 \ldots t_{j+1})|$. Since $ses(t_1 \ldots t_{j+1})$ is a suffix of $s_j t_{j+1}$, we obtain $s_{j+1} = ses(t_1 \ldots t_{j+1})$. Obviously, property 3 holds for V_{j+1}. If $s_{j+1} \in D_j$, then properties 4 and 5 hold for D_{j+1} and col_{j+1}. If $s_{j+1} \notin D_j$, then property 4 holds for D_{j+1}. Moreover, $col_{j+1}(s_{j+1}) = (f_{j+1}, g_{j+1}) \equiv (dcol(t_1 \ldots t_{j+1}), lcol(t_1 \ldots t_{j+1})) \equiv (dcol(s_{j+1}), lcol(s_{j+1}))$. □

Algorithm MDC computes a new pair (f_{j+1}, g_{j+1}) of columns only if $s_j t_{j+1} \notin V_j$. Thus, the total number of computed pairs of columns is bound by $q' = |\{s_j t_{j+1} \mid 0 \leq j \leq n-1\}|$. Moreover, a new pair of columns is stored only if $s_{j+1} \notin D_j$. Hence, the number of pairs of columns stored is $q = |D_n|$. Obviously, $q \leq q' \leq n$. A precise theoretical analysis of the parameters q and q' seems to be quite difficult for an arbitrary cost function. Boundary values for q and q' are only known for the unit cost model.

Lemma 6.2.14 [Ukk93a] If δ is the unit cost function, then $q = \mathcal{O}(\min\{n, m^{k+1} \cdot |\mathcal{A}|^k\})$ and $q' = \mathcal{O}(\min\{n, m^{k+1} \cdot |\mathcal{A}|^{k+1}\})$.

Proof See [Ukk93a]. □

In the following, we consider how to implement Algorithm MDC. In section 6.2.2, we outline the implementation techniques suggested in [Ukk93a]. These require to preprocess t into a compact suffix tree. In section 6.2.3, we present a new technique which does without preprocessing. The main advantage is that it uses atomic $\mathcal{A}^+$-trees and is therefore easier to implement than Ukkonen's techniques. Finally, section 6.2.4 is devoted to a new technique that connects the idea of Algorithm MDC with deterministic finite automata.

6.2.2 Ukkonen's Implementation Techniques for MDC

For ease of presentation, Ukkonen describes his implementation techniques for Algorithm MDC on the basis of an atomic $\mathcal{A}^+$-tree. In practice, a compact suffix trees is used instead. As an alternative, Ukkonen suggests to use a suffix automaton [BBH+85, Cro88] or a space economical suffix array [MM93a]. However, the latter data structure does not include suffix links (or something comparable). This may lead to problems.

Let us assume that $T = cst(t)$ including all inner suffix links has been preprocessed. To implement Algorithm MDC, each string $s \in V_j$ is represented in constant space by its location in T. col_j is stored in a hash table which maps $loc_T(s)$ to $col_j(s)$ for each $s \in D_j$. This table requires $\mathcal{O}(m \cdot q)$ space. Note that the hash table is necessary since $loc_T(s)$ can be an edge location of the form $(\overline{u}, v, w, \overline{uvw})$. An alternative solution is to create the node $\overline{uv}$ by splitting the edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ (see Definition 3.2.11). Then $col_j(s)$ can be stored at $\overline{uv}$ and a hash table is not necessary. However, the splitting changes the structure of T so that it cannot be used for solving an approximate string searching problem with a different pattern or threshold. Therefore, we will not consider the alternative solution.

To keep track of the locations representing V_j and D_j, one uses two additional hash tables. Both need $\mathcal{O}(n)$ space since $|V_j| \leq |V_n| \leq n$ and $|D_j| \leq |D_n| \leq n$. With this representation an implementation of Algorithm MDC should not be too difficult. The resulting program (cf. [Ukk93a, Algorithm A]) traverses T directed by the characters in t. Each input character t_{j+1} leads to a transition from $loc_T(s_j)$ to $loc_T(s_j t_{j+1})$. If $loc_T(s_j)$ is a node location, then such a transition may require to inspect $|\mathcal{A}|$ edges outgoing from $\overline{s_j}$. Starting with $loc_T(s_j t_{j+1})$, one successively calls the function *linkloc* until $loc_T(s_{j+1})$ is reached. The total number of calls to *linkloc* is bound by n and each call takes constant time on the average. Hence, the traversal requires $\mathcal{O}(|\mathcal{A}| \cdot n)$ steps. Additionally there are q' pairs of columns to be computed and q to be stored. Thus, the program achieves an overall running time of $\mathcal{O}(m \cdot q' + |\mathcal{A}| \cdot n)$ and uses $\mathcal{O}(m \cdot q + n)$ space.

One of the main contributions of Ukkonen is an implementation technique for Algorithm MDC that eliminates the dependency on n [Ukk93a, Algorithm B]. The technique works in two phases: First it computes all different shortest essential suffixes s_j and corresponding pairs (f_j, g_j) of columns, thereby skipping all steps of Algorithm MDC that do not produce a new shortest essential suffix. This is accomplished by using a dictionary H which supports operations for minimum extraction, deletion, and insertion. H is implemented by a balanced search tree. Thus, each operation can be performed in $\log |H|$ time. In a second phase, the compact suffix tree is traversed to obtain the solutions to the approximate string searching problem. In particular, a position j' is output whenever there is a shortest essential suffix s_j such that $f_j(m) \leq k$ and s_j ends at position j' in t. According to [Ukk93a, Theorem 10], this technique can be implemented in $\mathcal{O}(m \cdot q + n)$ time and $\mathcal{O}(m \cdot q)$ space if $|\mathcal{A}|$ is a constant.

Unfortunately, the second implementation technique requires considerable overhead. For this reason, Ukkonen devised a similar technique which does without a dictionary [Ukk93a, Algorithm C]. It is much easier to implement and runs fast in practice. However, it computes $\mathcal{O}(m \cdot q)$ extra pairs of columns. Therefore, the running time is $\mathcal{O}(m^2 \cdot q + \text{size of the output})$ and the space requirement is $\mathcal{O}(m^2 \cdot q)$ (see [Ukk93a, Theorem 11]).

6.2.3 An Online Implementation Technique for MDC

Ukkonen's implementation techniques lead to programs that traverse only a small subset of all locations in T, namely those locations representing V_n. In this section, we present a new technique, the online implementation technique. The idea is to use an atomic $\mathcal{A}^+$-tree and to construct only those parts of the tree that are likely to be traversed. In particular, for each $j, 0 \leq j \leq n$ one incrementally constructs the smallest atomic $\mathcal{A}^+$-tree T_j (including suffix links) such that $V_j \subseteq words(T_j)$ and $words(T_j)$ is suffix-closed. Obviously, T_0 is just the *root* with no edges. T_{j+1} is constructed from T_j as follows:

(*i*) If there is a t_{j+1}-edge outgoing from node $\overline{s_j}$, then $T_{j+1} = T_j$.

(*ii*) If there is no t_{j+1}-edge outgoing from node $\overline{s_j}$, then apply Algorithm 1 of [Ukk93b][2] with $top = \overline{s_j}$. This adds for each suffix s of s_j an edge $\overline{s} \xrightarrow{t_{j+1}} \overline{st_{j+1}}$ to T_j, until $\overline{s}$ is the *root* or such an edge already exists. Moreover, the algorithm adds a suffix link for $\overline{st_{j+1}}$ if necessary. The resulting atomic $\mathcal{A}^+$-tree is T_{j+1}.

[2] This algorithm is used for the online construction of atomic suffix trees.

The correctness of this construction is clear. According to [Ukk93a], a shortest essential suffix is of length $\mathcal{O}(m)$. Hence, case (ii) takes $\mathcal{O}(m \cdot |\mathcal{A}|)$ steps. Since it occurs q' times, T_n is constructed in $\mathcal{O}(m \cdot q' \cdot |\mathcal{A}|)$ time and $\mathcal{O}(m \cdot q')$ space.

Using T_j, we do not need hash tables. This considerably simplifies the implementation. V_j is represented by a boolean annotation of T_j. For each $s \in D_j$ the pair $col_j(s)$ of columns is stored at node $\overline{s}$ of T_j. A node at which no pair of columns is stored represents a string $s \notin D_j$. With this representation Algorithm MDC can be implemented in $\mathcal{O}(m \cdot q' \cdot |\mathcal{A}| + |\mathcal{A}| \cdot n)$ time and $\mathcal{O}(m \cdot q')$ space.

6.2.4 A Technique Based on Deterministic Finite Automata

The technique we describe in this section does not follow the control flow as specified in Algorithm MDC. Instead, it connects the ideas of Algorithm MDC with the concept of deterministic finite automata. The approach is to preprocess δ, $\mathcal{A}$, k, and p into a DFA whose states are the shortest essential suffixes. The DFA is constructed in a top down manner, in a similar way as the lazy suffix tree construction proceeds. Therefore, the construction phase of the algorithm can be interleaved with the enumeration of the protocol. This leads to an incremental construction which can easily be implemented in a lazy functional language.

The preprocessing technique is of special interest in contexts where the cost function, the input alphabet, the threshold, and the pattern are fixed, and the input string varies. Consider, for instance, the search for approximate occurrences of the "TATAAT"-box in DNA sequences.

Before we describe the preprocessing of the algorithm, we show some important properties of the set of all shortest essential suffixes.

Lemma 6.2.15 Let $SES = \{ses(v) \mid v \in \mathcal{A}^*\}$. SES is not empty, prefix-closed, and finite.

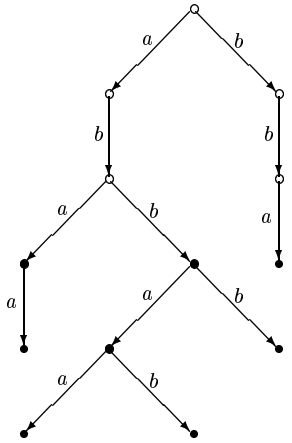
Proof Since $ses(\varepsilon) = \varepsilon$, we obtain $\varepsilon \in SES$. Hence, SES is not empty. Suppose $vb \in SES$. Then $ses(vb) = vb$. By Lemma 6.2.5, we obtain $ses(v)b \sqsubset ses(vb)$. Hence, $ses(v) \sqsubset v$. Since $v \sqsubset ses(v)$, we can conclude $ses(v) = v$, that is, $v \in SES$. Hence, SES is prefix-closed. As noted in [Ukk93a], a shortest essential suffix is of length $\mathcal{O}(m)$. Thus, there is a $q \geq 1$ such that $|s| \leq q \cdot m$ for each $s \in SES$. Therefore, SES is finite. □

According to Lemma 6.2.15, the set SES can be represented by an atomic $\mathcal{A}^+$-tree T such that $words(T) = SES$.

Example 6.2.16 Let δ be the unit cost function. Suppose $\mathcal{A} = \{a, b\}$, $k = 1$, and $p = abba$. Then SES is represented by the atomic $\mathcal{A}^+$-tree shown in Figure 6.2. Each node $\overline{s}$ such that $dcol(s)(m) \leq k$ is shown as a thick circle. □

Definition 6.2.17 The *SES-automaton* for δ, $\mathcal{A}$, k, and p is the deterministic finite automaton $(SES, \mathcal{F}, s_0, nextstate)$ where SES is as above, $\mathcal{F} = \{s \in SES \mid dcol(s)(m) \leq k\}$, $s_0 = \varepsilon$, and $nextstate(s, b)$ is the longest suffix of sb that occurs in SES. □

Figure 6.2: The Atomic $\mathcal{A}^+$-tree Representing *SES*



Example 6.2.18 Let δ, $\mathcal{A}$, k, and p be as in Example 6.2.16. The *SES*-automaton for δ, $\mathcal{A}$, p, and k is given by the following table. The accepting states are underlined. They correspond to the nodes in Figure 6.2 which are marked by thick circles.

	ε	a	ab	$\underline{aba}$	$\underline{abaa}$	$\underline{abb}$	$\underline{abba}$	$\underline{abbaa}$	$\underline{abbab}$	$\underline{abbb}$	b	bb	$\underline{bba}$
a	a	a	aba	$abaa$	a	$abba$	$abbaa$	a	aba	bba	a	bba	a
b	b	ab	abb	ab	ab	$abbb$	$abbab$	ab	abb	bb	bb	bb	ab

□

Algorithm SESA Construct the *SES*-automaton $M = (SES, \mathcal{F}, s_0, nextstate)$ for δ, $\mathcal{A}$, p, and k. Compute the protocol $(s_0, s_1, \ldots, s_n)$ of t w.r.t. M. Output j if $s_j \in \mathcal{F}$. □

Theorem 6.2.19 Algorithm SESA correctly solves the approximate string searching problem.

Proof Let $0 \leq j \leq n$. It suffices to prove that j is a solution to the approximate string searching problem if and only if $s_j \in \mathcal{F}$. In analogy to Theorem 5.2.2, one shows that s_j is the longest suffix of $t_1 \ldots t_j$ that occurs in *SES*. Let j be a solution to the approximate string searching problem. Then $dcol(t_1 \ldots t_j)(m) \leq k$. Obviously, we have $t_1 \ldots t_j \sqsupset s_j \sqsupset s$ where $s = ses(t_1 \ldots t_j)$. Hence, $dcol(s_j)(m) \leq dcol(s)(m) \leq k$ which implies $s_j \in \mathcal{F}$. Let $s_j \in \mathcal{F}$. Then $dcol(t_1 \ldots t_j)(m) \leq dcol(s_j)(m) \leq k$. Hence, j is a solution to the approximate string searching problem. □

The most interesting part of Algorithm SESA is the construction of the *SES*-automaton M. Lemma 6.2.15 suggests a construction which begins with state ε and obtains longer and longer states of *SES* in a top-town manner. Suppose $s \in SES$ and assume that we have computed $dcol(s)$ and $lcol(s)$. According to Lemma 6.2.8, $sb \in SES$ if and only if $|sb| = lcol(sb)(h)$ where $h = lei(dcol(sb))$. To check the latter condition, one evaluates the distance column $dcol(sb) = nextdcol(dcol(s), b)$ and the length column $lcol(sb) = nextlcol(dcol(s), lcol(s), b)$. If $|sb| = lcol(sb)(h)$, then one constructs a state sb and a transition $nextstate(s, b) = sb$. In this way, sb and a transition from s to sb are constructed in $\mathcal{O}(m)$ time.

Suppose $nextstate(s,b) \neq sb$. That is, a transition $nextstate(s,b) \neq sb$ is to be constructed. By definition, $nextstate(s,b)$ is the longest proper suffix of sb that occurs in SES. If $s = \varepsilon$, then we have $nextstate(s,b) = \varepsilon$. If $s \neq \varepsilon$, then $nextstate(s,b) = nextstate(s',b)$ where s' is the longest proper suffix of s that occurs in SES. The construction of the transitions can be organized such that state s' is available if $nextstate(s,b)$ is to be constructed (see section 6.2.5). Hence, $nextstate(s,b)$ can be found in constant time. This means that each transition $nextstate(s,b) \neq sb$ is constructed in $\mathcal{O}(m)$ time, due to the computation of $dcol(sb)$ and $lcol(sb)$.

Note that the top down approach allows to interleave the construction of M with the enumeration of the protocol of t w.r.t. M. Hence, M can be constructed incrementally. A transition is constructed immediately before it is to be evaluated for the first time. This is very similar to the lazy suffix tree algorithm. However, there are two important differences:

1. The construction of a transition $nextstate(s,b) \neq sb$ may require to construct a transition $nextstate(s',b) \neq s'b$. This may itself require to construct another transition, and so on.

2. It is necessary to store $dcol(s)$ and $lcol(s)$ whenever there is a character $b \in \mathcal{A}$ such that $nextstate(s,b)$ is not constructed yet. In the worst case, $\mathcal{O}(|SES| \cdot m)$ space is required for storing the columns.

The incremental construction can easily be implemented in a lazy functional language. This will be shown in section 6.2.5. In an eager language, the laziness can be simulated by explicit synchronization of the steps which construct a transition and the steps which evaluate a transition.

The most important disadvantage of the incremental construction is the worst case space requirement of $\mathcal{O}(|SES| \cdot m)$ for the columns. If one completely constructs M, the space requirement for the columns can considerably be reduced as shown in the following theorem.

Theorem 6.2.20 Algorithm SESA can be implemented in $\mathcal{O}(|SES| \cdot |\mathcal{A}| \cdot m + n)$ time and $\mathcal{O}(|SES| \cdot |\mathcal{A}| + m^2)$ space.

Proof In a first phase, the states and the transitions $nextstate(s,b) = sb$ can be constructed in a depth first strategy. This means that the states are created in lexicographic ordering. If s is the current state, then it suffices to store the distance and the length columns for the prefixes of s. Since $|s| \in \mathcal{O}(m)$, $\mathcal{O}(m^2)$ space is required for the columns. Hence, the first phase of the construction takes $\mathcal{O}(|SES| \cdot |\mathcal{A}| \cdot m)$ time and $\mathcal{O}(|SES| + m^2)$ space. In a second phase, the transitions $nextstate(s,b) \neq sb$ are constructed as described above. To guarantee that a transition is constructed before it is evaluated for the first time, one uses a breadth first strategy which visits state s' before state s, whenever $|s'| < |s|$. The second phase therefore takes $\mathcal{O}(|SES| \cdot |\mathcal{A}|)$ space and time. After M is constructed, the protocol of t w.r.t. M is enumerated in $\mathcal{O}(n)$ time, without using extra space. □

With the cutoff technique (see section 6.1), one can improve Algorithm SESA such that it takes $\mathcal{O}(|SES| \cdot |\mathcal{A}| \cdot k + n)$ time and $\mathcal{O}(|SES| \cdot |\mathcal{A}| + k \cdot m)$ space in the expected case. The same technique can obviously be applied in the incremental construction of the SES-automaton.

Now it remains to give an upper bound for the size of *SES*. As noted in [Ukk93a], the length of the shortest essential suffixes is $\mathcal{O}(m)$. Hence, there are at most $2 \cdot |\mathcal{A}|^{q \cdot m} - 1$ different shortest essential suffixes, for some $q \geq 1$. This result is not very satisfying. For a very general class of cost functions (which contains all integer cost functions) we can prove a useful property relating length and distance columns. This property can be exploited to derive a tighter upper bound.

Lemma 6.2.21 Suppose δ is a cost function such that $\delta(\varepsilon \to b) \geq 1$ for all insertion operations $\varepsilon \to b$. Then we have $lcol(v)(i) \leq i + dcol(v)(i)$ for each $v \in \mathcal{A}^*$ and each $i, 0 \leq i \leq m$.

Proof By simultaneous induction on i and v. If $i = 0$ or $v = \varepsilon$, then we have $lcol(v)(i) = 0 \leq i + dcol(v)(i)$. Let $f = dcol(vb)$ and $g = lcol(vb)$. To show $g(i+1) \leq i + 1 + f(i+1)$, we consider the following cases:

- If $f(i+1) = f(i) + \delta(p_{i+1} \to \varepsilon)$, then we can conclude $g(i+1) = g(i) \leq i + f(i) = i + f(i+1) - \delta(p_{i+1} \to \varepsilon) < i + f(i+1) < i + 1 + f(i+1)$.
- If $f(i+1) = dcol(v)(i) + \delta(p_{i+1} \to b)$, then $g(i+1) \leq lcol(v)(i) + 1 \leq i + dcol(v)(i) + 1 = i + 1 + f(i+1) - \delta(p_{i+1} \to b) \leq i + 1 + f(i+1)$.
- If $f(i+1) = dcol(v)(i+1) + \delta(\varepsilon \to b)$, then we have $g(i+1) \leq lcol(v)(i+1) + 1 \leq i + 1 + dcol(v)(i+1) + 1 = i + 1 + f(i+1) - \delta(\varepsilon \to b) + 1 \leq i + 1 + f(i+1)$.

Note that the restriction to the cost function is solely used to derive the last inequality. □

Theorem 6.2.22 Let δ be as in Lemma 6.2.21. Then $|SES| \leq 2 \cdot |\mathcal{A}|^{m+k} - 1$.

Proof Let $s \in SES$ and $h = lei(dcol(s))$. Then $|s| = lcol(s)(h) \leq h + dcol(s)(h)$, by Lemmas 6.2.8 and 6.2.21. Now $h \leq m$ and $dcol(s)(h) \leq k$. This implies $|s| \leq m + k$. Hence, there are at most $2 \cdot |\mathcal{A}|^{m+k} - 1$ different shortest essential suffixes. □

It is unclear whether the upper bound derived in Theorem 6.2.22 holds for arbitrary cost functions.

Before we present an implementation of Algorithm SESA, we briefly recall the main achievements of section 6.2. Based on the notion of essential suffixes, we have developed a declarative description of Algorithm MDC, including a detailed correctness proof. Algorithms A, B, and C of [Ukk93a] were described as special instances of Algorithm MDC. Moreover, we devised a new online implementation technique which is easier to implement since it is based on atomic $\mathcal{A}^+$-trees. Finally, we showed how to combine the ideas of Algorithm MDC with the concept of deterministic finite automata. An interesting subject of future work would be to compare all these technique on a practical basis.

6.2.5 Implementation

For the implementation of Algorithm SESA we need a function that simultaneously computes distance and length columns. As usual, we develop a corresponding table specification.

```
dltable::(costfunction α)→(string α)→tablespec α (num,num)
dltable delta p
  = (zip2 firstdcol (repeat 0),nextcol)
    where firstdcol = 0:[no + delta (D a) | (no,a)←zip2 firstdcol p]
          firstentry we b = (0,0)
          nextcol = absnextcol firstentry (dljoin3 delta) undef p

dljoin3::(costfunction α)→α→α→(num,num)→(num,num)→(num,num)→(num,num)
dljoin3 delta a b (dwe,lwe) (dno,lno) (dnw,lnw)
  = min2 (min2 (dwe + delta (I b),lwe+1) (dno + delta (D a),lno))
         (dnw + delta (R a b),lnw+1)
```

Let f be a distance column and g be a length column. f and g are represented by the list $[(f(0),g(0)),\ldots,(f(m),g(m))]$. *dltable* is a generalization of the table specification *seltable* (see section 6.1.1). While *seltable* computes the list *firstdcol* representing the distance column $dcol(\varepsilon)$, *dltable* computes the list $(zip2\ firstdcol\ (repeat\ 0))$ representing $dcol(\varepsilon)$ and $lcol(\varepsilon)$ (see Definition 6.2.9). The function *nextcol*, as specified in *dltable*, computes columns with the first entry $(0,0)$. *nextcol* uses a function $dljoin3$ which joins three pairs (dwe,lwe), (dno,lno), and (dnw,lnw) to yield a pair $(f(i+1),g(i+1))$. $f(i+1)$ is computed in the usual way by adding up corresponding costs and taking the minimum. $g(i+1)$ is simultaneously computed with $f(i+1)$ as the second component of some pair $(dwe+delta(I\,b),lwe+1)$, $(dno+delta(D\,a),lno)$, and $(dnw+delta(R\,a\,b),lnw+1)$, whose first component is $f(i+1)$. If there is more then one such pair, the property $lno \leq lnw+1 \leq lwe+1$ guarantees that $g(i+1)$ is correctly computed according to Definition 6.2.9. Note that *nextcol* corresponds to the function *dp* in [Ukk93a].

Based on the table specifications *dltable* and *selcotable*, one derives a cutoff version *dlcotable*.

```
dlcotable::(costfunction α)→num→(string α)→tablespec α (num,num)
dlcotable delta k p
  = (zip2 firstdcol (repeat 0),nextcol)
    where firstdcol = 0:takewhile (<=k) [no+delta (D a) | (no,a)←zip2 firstdcol p]
          firstentry we b = (0,0)
          dljoin2 a b (dno,lno) (dnw,lnw) s
            = takewhile ((<=k).fst) z
              where z = x:[(dno + delta (D a),lno) | ((dno,lno),a)←zip2 z s]
                        where x = min2 (dno+delta (D a),lno)
                                       (dnw+delta (R a b),lnw+1)
          nextcol f b
            = cutsuffix ((>k).fst) f'
              where f' = absnextcol firstentry (dljoin3 delta) dljoin2 p f b
```

In *dlcotable*, the columns f and g are represented by the list $[(f(0),g(0)),\ldots,(f(h),g(h))]$ where $h = lei(f)$. Note that $m = h$ if and only if $f(m) \leq k$. The function $dljoin3$ is reused to implement the function *nextcol* above. $dljoin2$ is a straightforward generalization of the function $join2$ in *selcotable*. Therefore, the correctness of *dlcotable* is obvious.

Suppose $\mathcal{A}$ is given as a pair $(characters,encode)$ of type $(alphabet\ \alpha)$. Let $l = |\mathcal{A}|$. The function *makeSESA* incrementally constructs the *SES*-automaton $M = (SES,\mathcal{F},s_0,nextstate)$ for δ, $\mathcal{A}$, p, and k. M is implemented by the expression $s0$ of type *dfa*. $s0$ represents the initial state s_0 (see Definition 3.8.5). Our implementation requires extension 1.

```
makeSESA::(costfunction α)→(alphabet α)→num→(string α)→dfa
makeSESA delta (characters,encode) k p
  = s0
    where m = #p
          l = #characters
          (firstcol,nextcol) = dlcotable delta k p
          blub = [(b,lookup (encode b)) | b←characters]
          s0 = S (m+1=#firstcol)
                 (makearray l [bsucc 1 (nextcol firstcol b) s0 | b←characters])
          bsucc d col (S accept' succ')
            = S (m=h) succ,     if d = snd (col!h)
            = S accept' succ',  otherwise
              where h = #col-1
                    succ = makearray l [bsucc (d+1) (nextcol col b)
                                                    (lub succ') | (b,lub)←blub]
```

Initially, *makeSESA* computes the table specification $(firstcol, nextcol)$ using the function *dlcotable*. Moreover, it constructs an ordered list *blub* of pairs (b, lub) for each character b. $lub = lookup\ (encode\ b)$ is a function that returns the entry at index $(encode\ b)$ if it is applied to some array. $s0$ is the expression $(S\ accept\ succ)$ where *accept* is true if and only if $dcol(\varepsilon)(m) \leq k$. *succ* is an array representing the successors of s_0. Note that we use programming with unknowns: $s0$ is used in the equation defining $s0$.

bsucc is the central function in *makeSESA*. Suppose $s \in SES$ and $b \in \mathcal{A}$. To compute the representation of the b-successor of s, *bsucc* is called with three arguments d, *col*, and $(S\ accept'\ succ')$ such that the following holds:

- $d = |sb|$,
- *col* represents $dcol(sb)$ and $lcol(sb)$, and
- $(S\ accept'\ succ')$ represents the longest proper suffix of sb that occurs in *SES*.

Note that $(S\ accept'\ succ')$ may not be constructed yet. However, due to the laziness we do not have to worry about this.

Let $h = |col| - 1$. Then h is the last essential index of $dcol(sb)$. Hence, if $d = snd\ (col!h)$ then $sb \in SES$ is the b-successor of s. sb is a new state which is represented by an expression $(S\ accept\ succ)$. *accept* is true if and only if $dcol(sb)(m) \leq k$. *succ* is an array representing the successors of sb. These are computed by recursively calling *bsucc* with the proper arguments for each $b \in \mathcal{A}$. If $d \neq snd(col!h)$, then the b-successor of s is the longest proper suffix of sb that occurs in *SES*. By assumption, the b-successor of s is represented by the expression $(S\ accept'\ succ')$.

It is easily verified that *makeSESA* computes each state and each transition of the *SES*-automaton M in $\mathcal{O}(m)$ time. Hence, our implementation is optimal.

Algorithm SESA is implemented by the function *appSESA*.

```
appSESA::(costfunction α)→(alphabet α)→num→(string α)→(string α)→[num]
appSESA delta alpha k p t = dfarun alpha (makeSESA delta alpha k p) t
```

Figure 6.3: Table D for $p = adbbc$ and $t = abbdadcbc$

		j									
	D	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
	1	1	0	1	1	1	0	1	1	1	1
i	2	2	1	1	2	1	1	0	1	2	2
	3	3	2	1	1	2	2	1	1	1	2
	4	4	3	2	1	2	3	2	2	1	2
	5	5	4	3	2	2	3	3	2	2	1

The construction phase is accomplished by *makeSESA*. For computing the solutions to the approximate string searching problem, we use the function *dfarun* which was introduced in section 3.8.1.

6.3 Properties of Table D

The k-differences (approximate string searching) problem is the approximate string searching problem restricted to the case of the unit cost function. For the rest of this chapter, we consider this problem. That is, we assume that k is a non-negative integer and that δ is the unit cost function.

Due to its interesting combinatorics, the k-differences problem has received a lot of attention [Ukk85b, LV88, LV89, GG88, GP90, JTU91, WMM91, Ukk92a, CL92, UW93, TU93, Mye94a, CL94, CM94]. Several fast algorithms have been developed. The key to most of these are the properties of the dynamic programming table D which are recalled in this section.

Definition 6.3.1 For each $i, 0 \leq i \leq m$ and each $j, 0 \leq j \leq n$ we define $D(i,j) = dcol(t_1 \ldots t_j)(i)$. A *$D$-diagonal* is a forward diagonal ($\searrow$) in D. More precisely, D-diagonal h consists of the values $D(i, h+i)$, $0 \leq i \leq m$, $0 \leq h+i \leq n$. $lastcol(h) = \min\{n, m+h\}$ is the last column on D-diagonal h. □

Example 6.3.2 [GP90] Let $k = 2$, $p = adbbc$, and $t = abbdadcbc$. Figure 6.3 shows the corresponding table D. D-diagonal 4 consists of the values 0, 0, 0, 1, 1, 1. The solutions to the k-differences problem are 3, 4, 7, 8, and 9. □

From Definitions 6.1.1 and 6.3.1 one derives the following properties of table D.

$$
\begin{aligned}
&D(0,0) = 0\\
&D(i+1,0) = D(i,0) + 1\\
&D(0,j+1) = 0\\
&D(i+1,j+1) = \min\left\{\begin{array}{l} D(i,j+1)+1\\ D(i+1,j)+1\\ D(i,j) + (\text{if } p_{i+1} = t_{j+1} \text{ then } 0 \text{ else } 1)\end{array}\right\}
\end{aligned}
$$

These recurrences are, for instance, stated in [Ukk85b, GG88, GP90, CL92, UW93, CL94]. Consecutive entries in D-columns, D-rows, and D-diagonals differ by at most one. Additionally the entries in D-diagonals are non-decreasing. This is formally stated in the following lemma.

Lemma 6.3.3 Table D has the following properties:

1. $D(i,j) - 1 \leq D(i+1,j) \leq D(i,j) + 1,\ 0 \leq i \leq m-1,\ 0 \leq j \leq n.$
2. $D(i,j) \leq D(i+1,j+1) \leq D(i,j) + 1,\ 0 \leq i \leq m-1,\ 0 \leq j \leq n-1.$
3. $D(i,j+1) - 1 \leq D(i,j) \leq D(i,j+1) + 1,\ 0 \leq i \leq m,\ 0 \leq j \leq n-1.$

Proof

1. Follows from Lemma 1 in [Ukk85b].
2. By [Ukk85b, Lemma 2], we have $D(i,j) \leq D(i+1,j+1)$. Moreover, by definition we have $D(i+1,j+1) \leq D(i,j) + \delta(p_{i+1} \to t_{j+1}) \leq D(i,j) + 1$.
3. For $i = 0$ the third inequality is obviously true. Suppose $i > 0$. Then by definition $D(i,j+1) \leq D(i,j) + 1$. Hence, we obtain $D(i,j+1) - 1 \leq D(i,j)$. Furthermore, $D(i,j) \leq D(i-1,j) + 1 \leq D(i,j+1) + 1$ by statement 1 and 2. □

Properties 1 and 2 were (essentially) first stated in [Ukk85b]. Note that all three properties hold for the table E_δ correspondingly. In fact, we have taken advantage of this in the table specification *uedisttable* for computing the unit edit distance fast (see section 3.9.5).

From the properties stated in Lemma 6.3.3 we can conclude the following:

Lemma 6.3.4 For all $i, j, 0 \leq i \leq m-1, 0 \leq j \leq n-1$ the following properties hold.

1. If $D(i,j) \leq D(i,j+1)$ and $D(i,j) \leq D(i+1,j)$, then $D(i,j) = D(i+1,j+1)$ if and only if $p_{i+1} = t_{j+1}$.
2. $$D(i+1,j+1) = \begin{cases} D(i,j), & \text{if } p_{i+1} = t_{j+1} \\ D(i,j+1) + 1, & \text{else if } D(i,j+1) < D(i,j) \\ 1 + \min\{D(i+1,j), D(i,j)\}, & \text{otherwise} \end{cases}$$

Proof

1. By assumption, we have
$$D(i+1,j+1) = \min \left\{ \begin{array}{l} D(i+1,j) + 1, \\ D(i,j+1) + 1, \\ D(i,j) + \delta(p_{i+1} \to t_{j+1}) \end{array} \right\} = D(i,j) + \delta(p_{i+1} \to t_{j+1})$$
Hence, $D(i,j) = D(i+1,j+1) \iff \delta(p_{i+1} \to t_{j+1}) = 0 \iff p_{i+1} = t_{j+1}$.
2. Routine. □

In the following sections, we consider the most important algorithms for solving the k-differences problem. Except for the algorithms described in section 6.8, all take advantage of the properties stated in Lemmas 6.3.3 and 6.3.4.

6.4 Ukkonen's Cutoff Algorithm

The algorithm considered in this section is Algorithm SELco (see section 6.1) restricted to the unit cost function. This restriction allows some optimizations. Since the algorithm is due to Ukkonen [Ukk85b], we call it UKKco. Jokinen et al. [JTU91] instead use the term "enhanced dynamic programming".

Algorithm UKKco [Ukk85b] Compute $D(i,0) = i$ for each $i, 0 \leq i \leq k$, and set $l_0 = k$. For each $j, 0 \leq j \leq n-1$ and each $i, 0 \leq i \leq l_j$, perform the following steps:

(1) If $i < l_j$, then compute $D(i+1, j+1)$ according to Lemma 6.3.4, Statement 2.

(2) Let $i = l_j < m$. If $p_{i+1} = t_{j+1}$ or $k > D(i, j+1)$, then set $D(i+1, j+1) = k$ and $l_{j+1} = l_j + 1$. Otherwise, set $l_{j+1} = \max\{l \mid 0 \leq l \leq l_j, D(l, j+1) = k\}$.

For each $j, 0 \leq j \leq n$ output j if $l_j = m$. □

Theorem 6.4.1 Algorithm UKKco correctly solves the k-differences problem.

Proof We show by induction on j that l_j is the last essential index of the j-th column of table D, and that the values $D(i,j)$ are computed correctly for each $i, 0 \leq i \leq l_j$. This implies the correctness. For $j = 0$ the claim easily follows. Suppose the claim holds for some $j, 0 \leq j \leq n-1$. Let $0 \leq i \leq l_j$.

(1) If $i < l_j$, then $D(i+1, j+1)$ depends on $D(i, j+1)$, $D(i+1, j)$ and $D(i,j)$. These values are computed correctly. Hence, $D(i+1, j+1)$ is computed correctly according to Lemma 6.3.4, Statement 2.

(2) Let $i = l_j < m$. Then $D(i,j) = k$ and $k \leq D(i+1, j+1) < D(i+2, j+1) < \cdots < D(m, j+1)$. If $p_{i+1} = t_{j+1}$ or $k > D(i, j+1)$, then $D(i+1, j+1) = k$ and $l_{j+1} = l_j + 1$ is the last essential index of column $j+1$. Otherwise, $D(i+1, j+1) > k$ and the last essential index of the column $j+1$ is $\leq l_j$. □

Algorithm UKKco computes $\mathcal{O}(l_j)$ entries in column j of table D. In the worst case, $l_j = m$. Hence, the worst case running time is $\mathcal{O}(m \cdot n)$. Ukkonen [Ukk85b] stated that the average value of the l_j is $\mathcal{O}(k)$. Chang and Lampe [CL92] provided the first proof of this statement. Thus, we can conclude that the expected running time of Algorithm UKKco is $\mathcal{O}(k \cdot n)$.

6.4.1 Implementation

From the table specification for Algorithm SELco we can easily derive a table specification for Algorithm UKKco.

```
ukkcutofftable::num→(string α)→tablespec α num
ukkcutofftable k p
  = (firstcol,nextdcol)
    where firstcol = [0..k]
          firstentry we b = 0
          join2 a b no nw s = [k], if a = b ∨ k > no
                            = [],  otherwise
          nextdcol f b = cutsuffix (>k) f'
                       where f' = absnextcol firstentry ujoin3 join2 p f b
```

In the first column of table D, only $k+1$ entries are computed. These are represented by the list $[0 \ldots k]$. Case (1) of Algorithm UKKco is implemented by the function $ujoin3$ which was already used for computing the unit edit distance (see section 3.9.5). Case (2) is implemented by the function $join2$. $ujoin3$ and $join2$ serve as the second and third argument of the function *absnextcol*. Note that if $join2$ computes the empty list, the column f' returned by *absnextcol* may end with a non-empty suffix of values that are greater than k. This suffix is cut off in $\mathcal{O}(|f'|)$ steps using the function (*cutsuffix* $(> k)$).

appUKKco implements UKKco. Note that *appUKKco* differs from *appSELco* only in the table specification.

```
appUKKco::num→(string α)→(string α)→[num]
appUKKco k p t = [j | (j,f)←zip2 [0..] (scanl nextdcol firstcol t); ok f]
                 where (firstcol,nextdcol) = ukkcutofftable k p
                       ok f = [] ~= drop (#p) f
```

From the above it is clear that *appUKKco* is an optimal implementation of Algorithm UKKco.

6.5 Ukkonen's Column-DFA

In [Ukk85b], an algorithm is described that preprocesses $\mathcal{A}$, k, and p into a deterministic finite automaton that scans the input string t in $\mathcal{O}(n)$ steps, thereby finding all approximate matches. Intuitively, each state of the automaton represents a possible column of table D and each transition represents the computation of a column from a previous column. Recall that Algorithm SESA also constructs a DFA. However, unlike Algorithm SESA, Ukkonen's algorithm is restricted to the unit cost function.

Let f be a distance column. If $f(i) > k$, then the *exact* value of $f(i)$ does not matter (see Lemma 6.1.4). Hence, each value $f(i)$ larger than k can be set to $k+1$, without affecting the correctness of Sellers' algorithm. This motivates the following definition.

Definition 6.5.1 For each $v \in \mathcal{A}^*$ the column $ndcol(v)$ is defined as follows:

$$ndcol(v)(i) = \min\{dcol(v)(i), k+1\}$$

for all $i, 0 \leq i \leq m$. $ndcol(v)$ is the *normalized distance column* of v. $f \in C$ is a normalized distance column if $f = ndcol(v)$ for some $v \in \mathcal{A}^*$. □

Note that a normalized distance column contains $m+1$ values, all of which are $\leq k+1$. Hence, there are only finitely many normalized distance columns. Ukkonen defines his automaton by giving a program for its construction. Our column-notation allows a very convenient declarative definition.

Definition 6.5.2 The *column*-DFA for $\mathcal{A}$, k, and p is the deterministic finite automaton $(\mathcal{S}, \mathcal{F}, s_0, nextndcol)$ which is specified as follows:

1. $\mathcal{S} = \{ndcol(v) \mid v \in \mathcal{A}^*\}$,
2. $\mathcal{F} = \{f \in \mathcal{S} \mid f(m) \leq k\}$,
3. $s_0 = ndcol(\varepsilon)$,
4. $nextndcol(ndcol(v), b) = ndcol(vb)$, for all $v \in \mathcal{A}^*$ and all $b \in \mathcal{A}$. □

Notice that one can define the column-DFA on *all* distance columns. (Ukkonen initially did this in [Ukk85b].) However, using normalized distance columns reduces the size of the automaton considerably.

Example 6.5.3 Let $\mathcal{A} = \{a, b\}$, $k = 1$, and $p = abba$ as in Example 6.2.16. The column-DFA for $\mathcal{A}$, k, and p is represented by the following table. The states are written as sequences of integers. The accepting states are underlined.

	01222	01122	01112	<u>01111</u>	<u>01101</u>	01012	<u>01011</u>	00122	<u>00121</u>	<u>00111</u>	<u>00110</u>
a	00122	00122	00121	00121	00110	00111	00111	00122	00122	00121	00121
b	01122	01112	01112	01112	01111	01101	01101	01012	01012	01012	01011

The automaton has 11 states. Note that the corresponding *SES*-automaton as shown in Example 6.2.18 has 13 states. This is because $ndcol(abaa) = ndcol(abbaa) = ndcol(bba) =$ 00121, that is, the shortest essential suffixes *abaa*, *abbaa*, and *bba* map to the same normalized distance column 00121. All other shortest essential suffixes map to a different column. □

Note that for each state $f \in \mathcal{S}$ there is at least one shortest essential suffix s such that $ndcol(s) = f$. This means that the column-DFA never has more states than the corresponding *SES*-automaton. In particular, the column-DFA can be considered as a minimized *SES*-automaton, in which shortest essential suffixes are identified if they yield the same normalized distance column.

Algorithm Cdfa [Ukk85b] Preprocess the column-DFA $M = (\mathcal{S}, \mathcal{F}, s_0, nextndcol)$ for $\mathcal{A}$, k, and p. Compute the protocol $(s_0, s_1, \ldots, s_n)$ of t w.r.t. M. Output j if $s_j \in \mathcal{F}$. □

The crucial point in Algorithm Cdfa is the preprocessing phase. Ukkonen [Ukk85b] gives an imperative program for constructing a column-DFA. By an iteration over $\mathcal{A}$ and an already computed subset of $\mathcal{S}$, the program successively constructs the remaining states, together with the transitions. In order to compute a transition efficiently, the representation of $\mathcal{S}$ is very important. Ukkonen suggests to represent each $f \in \mathcal{S}$ by the sequence

$$(f(1) - f(0), f(2) - f(1), \ldots, f(m) - f(m-1)) \tag{6.1}$$

of differences. According to Lemma 6.3.3, Statement 1, $f(i) - f(i-1) \in \{1, 0, -1\}$ for all $i, 1 \leq i \leq m$. Hence, (6.1) can conveniently be stored in a *ternary tree* whose edge labels are 1, 0, and -1. Each membership test and each insertion into the ternary tree can be performed in $\mathcal{O}(m)$ steps. Thus, a transition can be computed in $\mathcal{O}(m)$ steps. Let $l = |\mathcal{A}|$. Since there are $|\mathcal{S}| \cdot l$ transitions, the column-DFA can be constructed in $\mathcal{O}(|\mathcal{S}| \cdot l \cdot m)$ time. The space requirement for the ternary tree is $\mathcal{O}(|\mathcal{S}| \cdot m)$.

By the cutoff technique described in section 6.4, one can improve the average case efficiency of the preprocessing.[3] In particular, each $f \in \mathcal{S}$ can be uniquely represented by the sequence

$$(f(1) - f(0), f(2) - f(1), \ldots, f(lei(f)) - f(lei(f) - 1)) \tag{6.2}$$

Since the expected value of the last essential index is $\mathcal{O}(k)$ (see [CL92]), a membership test and an insertion into the ternary tree takes $\mathcal{O}(k)$ steps in the expected case. Hence, the construction time and the size of the ternary tree reduce in the expected case to $\mathcal{O}(|\mathcal{S}| \cdot l \cdot k)$ and $\mathcal{O}(|\mathcal{S}| \cdot k)$, respectively.

In every step of the preprocessing phase, the new states, that is, those for which the transitions have not been computed yet, must be stored. As all new states occur in the ternary tree, Ukkonen suggests to use a *queue* of pointers to the nodes representing the new states. This queue takes $\mathcal{O}(|\mathcal{S}|)$ space, and deletions and insertions can be performed in constant time. Hence, the worst case preprocessing time is $\mathcal{O}(|\mathcal{S}| \cdot l \cdot m)$ and the space requirement is $\mathcal{O}(|\mathcal{S}| \cdot (l + m))$. To obtain the complexities for the average case, one substitutes m by k.

The representation of $\mathcal{S}$ by a ternary tree implies $|\mathcal{S}| \in \mathcal{O}(3^m)$. Taking the threshold and the size of the input alphabet into consideration, Ukkonen derives a second upper bound. He shows that $|\mathcal{S}| \in \mathcal{O}(2^k \cdot l^k \cdot m^{k+1})$. This gives the following result.

Theorem 6.5.4 [Ukk85b] Algorithm Cdfa correctly solves the k-differences problem in $\mathcal{O}(q \cdot l \cdot m + n)$ time and $\mathcal{O}(q \cdot (l + m))$ space where $q = \min\{3^m, 2^k \cdot l^k \cdot m^{k+1}\}$ and $l = |\mathcal{A}|$.

Proof See [Ukk85b]. □

Theorem 6.5.4 shows that if m and k are not quite small, the large time and space requirements may limit the applicability of Algorithm Cdfa. However, there are some practical improvements:

- One can organize the algorithm such that the construction of the column-DFA is interleaved with the computation of the protocol. That is, a transition is constructed immediately before it is evaluated for the first time. Such an incremental construction may lead to an improvement of the overall running time. It can easily be implemented in a lazy language, as we will show in the following section.

- Ukkonen [Ukk85b] suggested to identify normalized distance columns if the first, say, $(3 \cdot k)/2$ entries are equal. This considerably reduces the number of states and the preprocessing effort. Of course, simultaneously the search phase slows down since the entries $f(i) \leq k$ for each $i, 3 \cdot k/2 < i \leq m$ must be maintained, while the input string is scanned. However, on the average such entries do not occur, and the search phase still take $\mathcal{O}(n)$ time in the expected case.

[3] In fact, Ukkonen described the cutoff technique in his paper [Ukk85b] about the column-DFA.

- Wu, Manber, and Myers [WMM92] have carried Ukkonen's idea further. They describe an algorithm which directly maintains the list of differences. That is, instead of computing $f' = nextndcol(f, b)$, some dynamic programming on the list (6.2) is performed to obtain the list $(f'(1) - f'(0), f'(2) - f'(1), \ldots, f'(lei(f')) - f'(lei(f') - 1))$. This means that the normalized distance columns do not explicitly occur in the algorithm. The main idea of Wu, Manber, and Myers is to partition the lists of differences into non-overlapping regions of length r where $r \in \mathcal{O}(\log n)$. Since the list of differences is of expected length $\mathcal{O}(k)$, there are $\mathcal{O}(k/r)$ such regions. Each region is uniquely encoded as a state of a "universal" automaton which is used to compute one region from a previous region in constant time. In this way, a state transition of the column-DFA is simulated by $\mathcal{O}(k/r)$ state transitions of the universal automaton. The latter automaton is much smaller than the former. In particular, Wu, Manber, and Myers show that the universal automaton requires $\mathcal{O}(|\mathcal{A}| \cdot m/\log n + n)$ space and that it can be computed in $\mathcal{O}(|\mathcal{A}| \cdot m + n)$ time. Now consider the search phase. The universal automaton performs $\mathcal{O}(k \cdot n/\log n)$ state transitions, each taking constant time. Thus, the search phase takes $\mathcal{O}(k \cdot n/\log n)$ expected time. Wu, Manber, and Myers have shown that in practice their algorithm is four to five times faster than Algorithm UKKco. They present no measurements which compare their algorithm to Algorithm Cdfa.

6.5.1 Implementation

In this section, we first consider the main data types used for implementing Algorithm Cdfa. Then we give a greedy preprocessing function that always constructs the column-DFA completely. Moreover, we develop a function that performs an incremental construction, thereby exploiting lazy evaluation. Finally, we combine the preprocessing phase with the search phase.

Let $(\mathcal{S}, \mathcal{F}, s_0, nextndcol)$ be the column-DFA for $\mathcal{A}$, k, and p. A state $f \in \mathcal{S}$ is represented by the list $[f(0), f(1), \ldots, f(lei(f))]$. Hence, f is an accepting state if and only if this list is of length $m + 1$. If we apply the following function *diffs* to the above list, then we obtain in $\mathcal{O}(lei(f))$ steps the list (6.2) of differences. Notice that $(scanl\ (+)\ 0)$ is the inverse function to *diffs*.

```
diffs::[num]→[num]
diffs f = [e-e' | (e,e')←zip2 (drop 1 f) f]
```

A ternary tree is implemented by an expression of type $(tern\ \alpha)$. Each node is labeled by an expression of type $(value\ \alpha)$.

```
tern α ::= T (value α) (tern α) (tern α) (tern α) | Emptytern
value α ::= Defval α | Undefval
```

A non-empty ternary tree is implemented by an expression of the form $(T\ v\ e\ e'\ e'')$. An implicit edge labeled 1 leads to the subtree e. An implicit edge labeled 0 leads to the subtree e'. An implicit edge labeled -1 leads to the subtree e''. Let *tern* be an expression of type $(tern\ dfa)$. A node in *tern* is denoted by $\overline{v}$ if v is the path from the *root* to the node. The

state $f \in \mathcal{S}$ occurs in *tern* if there is a node $\overline{v}$ such (*diffs* $f = v$) and $\overline{v}$ is labeled by an expression (*Defval* s) where the expression s of type *dfa* represents f according to Definition 3.8.5. The constructor *Undefval* is used for labeling nodes which do not represent a state. We use the function *getvalue* to select an α-value from an expression of the form (*Defval* s).

```
getvalue::(value α)→α
getvalue (Defval s) = s
```

Note that we could also use the data type *tree* to implement a ternary tree. However, this would lead to a tree structure with explicit edge labels, which would nearly double the size of the representation.

Let ds be a list of differences 0, 1, and -1. (*newtern* s ds) returns in $\mathcal{O}(|ds|)$ time a ternary tree with the path ds. The node corresponding to ds is labeled with (*Defval* s).

```
newtern::α→[num]→(tern α)
newtern s (1:ds) = T Undefval (newtern s ds) Emptytern Emptytern
newtern s (0:ds) = T Undefval Emptytern (newtern s ds) Emptytern
newtern s (-1:ds) = T Undefval Emptytern Emptytern (newtern s ds)
newtern s [] = T (Defval s) Emptytern Emptytern Emptytern
```

(*lookuptern* ds *tern*) looks for the path ds in the ternary tree *tern*. If the path exists, then the label of the corresponding node is returned. Otherwise, *Undefval* is returned. The running time is $\mathcal{O}(|ds|)$.

```
lookuptern::[num]→(tern α)→value α
lookuptern (1:ds) (T value e e' e'') = lookuptern ds e
lookuptern (0:ds) (T value e e' e'') = lookuptern ds e'
lookuptern (-1:ds) (T value e e' e'') = lookuptern ds e''
lookuptern [] (T value e e' e'') = value
lookuptern ds Emptytern = Undefval
```

(*inserttern* s ds *tern*) updates the ternary tree *tern*. That is, it constructs a node corresponding to the path ds if such a node does not already exist in *tern*. The node is labeled with (*Defval* s). The running time is $\mathcal{O}(|ds|)$.

```
inserttern::α→[num]→(tern α)→(tern α)
inserttern s (1:ds) (T value e e' e'') = T value (inserttern s ds e) e' e''
inserttern s (0:ds) (T value e e' e'') = T value e (inserttern s ds e') e''
inserttern s (-1:ds) (T value e e' e'') = T value e e' (inserttern s ds e'')
inserttern s [] (T value e e' e'') = T (Defval s) e e' e''
inserttern s ds Emptytern = newtern s ds
```

We assume that $\mathcal{A}$ is given as a pair (*characters*, *encode*) of type (*alphabet* α). The function *makecdfa* computes the column-DFA for $\mathcal{A}$, k, and p.

```
makecdfa::[α]→num→(string α)→dfa
makecdfa characters k p
  = s0
    where m = #p
          l = #characters
          (firstcol,nextdcol) = ukkcutofftable k p
          nextndcol f b = map (min2 (k+1)) (nextdcol f b)
          initialtern = inserttern s0 (diffs firstcol) Emptytern
          s0 = S (m=k) (makearray l (fst (gettrans firstcol initialtern)))
          gettrans f tern = foldr bsucc ([],tern) (map (nextndcol f) characters)
          bsucc f' (ss,tern)
            = (s:ss,tern'),              if value = Undefval
            = (getvalue value:ss,tern), otherwise
              where df' = diffs f'
                    value = lookuptern df' tern
                    s = S (1+m=#f') (makearray l translist')
                    (translist',tern') = gettrans f' (inserttern s df' tern)
```

Initially, *makecdfa* computes m and l and calls the function *ukkcutofftable* (see section 6.4) to obtain the table specification $(firstcol, nextdcol)$. By definition, *firstcol* represents $ndcol(\varepsilon)$. The function *nextndcol* is easily implemented by combining *nextdcol* with a normalization step accomplished by the function $(map\ (min2\ (k+1)))$. The running time of *nextndcol* is proportional to the length of f. According to Definition 3.8.5, the column-DFA is implemented by an expression $s0$ of type *dfa*. The successors of $s0$ are computed by evaluating the expression $(gettrans\ firstcol\ initialtern)$ where *initialtern* is a ternary tree in which *firstcol* occurs as the only state. *initialtern* is obtained by inserting $s0$ with the path $(diffs\ firstcol)$ into an empty ternary tree. Note that we use programming with unknowns: $s0$ depends on *initialtern* and vice versa. However, since we do not need to evaluate $s0$ to compute *initialtern* the mutual dependency does not lead to problems.

Let $f \in \mathcal{S}$ and suppose *tern* is a ternary tree representing some subset of $\mathcal{S} \setminus \{f\}$. The expression $(gettrans\ f\ tern)$ returns a pair $(translist, tern')$. *translist* is the list of successors of f. $tern'$ is a ternary tree in which all states occur that have been obtained while computing *translist*. *gettrans* is implemented by iterating the function *bsucc* on the states $(f' = nextndcol\ f\ b)$, for each $b \in \mathcal{A}$. *bsucc* works as follows. It computes the list df' of differences for f' and looks up df' in *tern*. This takes $\mathcal{O}(|f'|)$ steps and gives an expression *value* of type $(value\ dfa)$. Consider the following cases.

- If $value = Undefval$, then f' does not occur in *tern*. Therefore, an expression s is computed which represents f'. s is inserted in *tern*. This takes $\mathcal{O}(|f'|)$ steps and gives a ternary tree to which *gettrans* is applied in order to compute the successors of s and the updated ternary tree $tern'$. If there are h states occurring in *tern* and h' states occurring in $tern'$, then the running time of *gettrans* is $\mathcal{O}(|h' - h| \cdot l \cdot r)$ where $l = |\mathcal{A}|$ and r is the average length of the $r' - r$ states inserted in *tern*. Note again the use of programming with unknowns: s depends on $translist'$ and vice versa. Since s can be inserted into *tern* without evaluating s, this dependency does not cause an error.

- If $value \neq Undefval$, then f' occurs in *tern* and the b-successor of f is selected from the expression *value*.

From the remarks above, it is clear that the running time of *makecdfa* is $\mathcal{O}(|\mathcal{S}| \cdot l \cdot m)$ in the worst case and $\mathcal{O}(|\mathcal{S}| \cdot l \cdot k)$ in the average case. Note that the function *makecdfa* computes the column-DFA in a greedy strategy. Whenever a new state f' is obtained, the function *bsucc* immediately constructs a path ($v = diffs\ f'$) in the actual ternary tree and labels the node $\overline{v}$ with the *dfa*-expression s representing f'. Due to the laziness, it is not necessary to evaluate s. However, after ($bsucc\ f'\ (ss, tern)$) is evaluated, other calls of the function *bsucc* may occur which look up or insert some value in $tern'$. To perform these operations, all ternary operation initiated by the call ($bsucc\ f'\ (ss, tern)$) must be completed. Hence, *makecdfa* always evaluates the column-DFA completely. The construction is not incremental which means that it is not very useful in practice. However, we have included it here because it is an optimal implementation which is simple and elegant due to the use of programming with unknowns.

An alternative to the greedy strategy is to construct a path ($v = diffs\ f'$) in the actual ternary tree and to label the node $\overline{v}$ with a unique number, the state number of f'. f' is stored in a queue. In this way, the computation of the *dfa*-expression representing f' is postponed until all states with a smaller state number have been completely processed. In the following, we show how to implement the postponing strategy. We need a data type ($queue\ \alpha$) supporting the following four functions:

```
emptyqueue::queue α
queueisempty::(queue α)→bool
enqueue::(queue α)→α→(queue α)
dequeue::(queue α)→(α,queue α)
```

emptyqueue returns the empty queue. ($queueisempty\ q$) checks if q is the empty queue. ($enqueue\ q\ a$) adds an element a to the back of q. *dequeue* is applied to a non-empty queue q and returns a pair (a, q') where a is the first element of the front of q and q' is q after removing a from the front. In Appendix A, we show how to implement queues purely functionally such that the four operations take constant time on the average.

Let $characters = [c_0, \ldots, c_{l-1}]$. The function *makeftrans* constructs a list

$$ftrans = [(f_0, trans_0), \ldots, (f_h, trans_h)]$$

such that the following is true: $\mathcal{S} = \{f_0, \ldots, f_h\}$, and for each $j, 0 \leq j \leq h$, $trans_j$ is the list $[i_0, \ldots, i_{l-1}]$ of indices where $nextndcol(f_j, c_r) = f_{i_r}$ for each $r, 0 \leq r \leq l-1$. Obviously, $ftrans$ is a representation of the column-DFA.

```
makeftrans::[α]→num→(string α)→[([num],[num])]
makeftrans characters k p
  = generate (newtern 0 (diffs firstcol)) 1 (enqueue emptyqueue firstcol)
    where (firstcol,nextdcol) = ukkcutofftable k p
          nextndcol f b = map (min2 (k+1)) (nextdcol f b)
          generate tern snr new
            = [],                                  if queueisempty new
            = (f,trans):generate tern' snr' new', otherwise
              where (f,new'') = dequeue new
                    (trans,tern',snr',new') = foldr bsucc' ([],tern,snr,new'')
                                                (map (nextndcol f) characters)
```

makeftrans uses a function *generate* to construct *ftrans* from left to right. *generate* has three arguments: a ternary tree *tern*, an integer *snr*, and a queue *new*. Suppose *generate* has constructed $[(f_0, trans_0), \ldots, (f_{h'}, trans_{h'})]$ for some $h' \leq h$. Then *new* contains the states $f_{h'+1}, \ldots, f_{snr-1}$. Moreover, *tern* represents the states $f_0, \ldots, f_{snr-1}$, and a node in *tern* which corresponds to a state f_r is labeled by the expression $(Defval\ r)$. Note that r is of type *num*. Due to its polymorphism, we can reuse the data type $(tern\ \alpha)$ and instantiate α by *num*. *generate* works as follows. If *new* is empty, then $h' = h$ and *ftrans* is computed. If *new* is not empty, then $f = f_{h'+1}$ is dequeued which gives a queue new''. To obtain the list $trans = trans_{h'+1}$ and appropriately updated values $tern'$, snr' and new', the function $bsucc'$ is iteratively applied to the states $(f' = nextndcol\ f\ b)$, for each $b \in \mathcal{A}$.

```
bsucc'::[num]→([num],tern num,num,queue [num])→([num],tern num,num,queue [num])
bsucc' f' (ss,tern,snr,new)
  = (snr:ss,tern',snr+1,enqueue new f'), if value = Undefval
  = (getvalue value:ss,tern,snr,new),    otherwise
    where df' = diffs f'
          value = lookuptern df' tern
          tern' = inserttern snr df' tern
```

$bsucc'$ computes the list df' of differences for f' and looks up df' in *tern*. This gives an expression *value* of type $(value\ num)$. Consider the following cases.

- If $value = Undefval$, then f' does not occur in *tern*. Hence, *snr* is the state number of f'. *tern* is updated such that it contains f' with the state number *snr*. *snr* in incremented, and f' is enqueued. This means that the computation of the successors of f' is postponed until all states which are already in the queue have been completely processed.
- If $value \neq Undefval$, then f' occurs in *tern*. The state number of f' is selected from the expression *value*.

$bsucc'$ needs $\mathcal{O}(|f'|)$ steps. Hence, the running time of *makeftrans* is $\mathcal{O}(|\mathcal{S}| \cdot l \cdot m)$ in the worst case and $\mathcal{O}(|\mathcal{S}| \cdot l \cdot k)$ in the average case.

The function $makecdfa'$ constructs the column-DFA using the postponing strategy. It computes the list *ftrans* and transforms each pair $(f, trans)$ into an expression $(S\ accept\ succ)$ representing f. *accept* is true if and only if $f(m) \leq k$. For each i_r in *trans* the entry $succ[r]$ in the array *succ* is obtained by selecting the element with index i_r in the list *dfastates*. Note the use of programming with unknowns. The list *dfastates* is constructed and looked up simultaneously.

```
makecdfa'::[α]→num→(string α)→dfa
makecdfa' characters k p
  = hd dfastates
    where m = #p
          l = #characters
          ftrans = makeftrans characters k p
          dfastates = map transform ftrans
          transform (f,trans) = S (1+m=#f) (makearray l [dfastates!i | i←trans])
```

appCdfa implements Algorithm Cdfa. The preprocessing is accomplished by *makecdfa'*. Searching is done by the function *dfarun*.

```
appCdfa::(alphabet α)→num→(string α)→(string α)→[num]
appCdfa alpha k p t = dfarun alpha (makecdfa' (fst alpha) k p) t
```

makecdfa' performs $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{A}|)$ accesses to the list *dfastates* using the predefined operator (!). Each access takes $\mathcal{O}(|\mathcal{S}|)$ steps on the average. Hence, the transformation of *ftrans* into the column-DFA requires $\mathcal{O}(|\mathcal{S}|^2 \cdot |\mathcal{A}|)$ time if the list *dfastates* is completely evaluated. However, *makecdfa'* constructs the column-DFA incrementally. Only those elements of the list *dfastates* are computed which occur in the protocol of t w.r.t. the column-DFA. Moreover, there are at most n application of the operator (!). So, if the column-DFA is large and n is not too large, the postponing strategy may lead to an improved overall running time.

Note that if we replace *makecdfa'* by *makecdfa*, we obtain an optimal implementation of Algorithm Cdfa. This, however, has the undesirable property that it always constructs the column-DFA completely. It is unclear how to obtain an optimal purely functional implementation that performs an incremental construction of the column-DFA. It should be remarked that it does not make sense to transform the list *dfastates* into an array in order to have constant time access to each state. For creating an array, its size must be known. This can only be determined by computing the length of *ftrans* which requires the construction of all states of the column-DFA. This, of course, contradicts the incremental construction.

6.6 Diagonal Transition Algorithms

The values along a D-diagonal are non-decreasing and increase only in unit steps. Hence, a D-diagonal may contain consecutive identical values. The basic idea of the Diagonal Transition Algorithms is to omit the calculation of identical values and instead compute only *diagonal transitions*, that is, positions where the diagonal values increase.[4] The first $k+1$ transitions along each diagonal are sufficient to characterize the solutions to the k-differences problem. In order to compute a diagonal transition, it is necessary to determine the "jump" of a suffix of p and a suffix of t.

Definition 6.6.1 We define a function $jump : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{N}_0$ by

$$jump(x, y) = \max\{|z| \mid z \sqsubset x, z \sqsubset y\}.$$

That is, $jump(x, y)$ is the length of the longest common prefix of x and y. $jump(x, y)$ is called the jump for x and y. □

The various Diagonal Transition Algorithms only differ in the way jumps are computed. A brute force algorithm directly compares characters of the pattern and of the input string. It achieves a worst case running time of $\mathcal{O}(m \cdot n)$ and an expected running time of $\mathcal{O}(k \cdot n)$. Alternative algorithms preprocess the pattern and the input string in order to compute jumps in constant time, thereby improving the worst case behavior to $\mathcal{O}(k \cdot n)$. The following results have been obtained for preprocessing.

[4] The term "diagonal transition" is adopted from Chang and Lampe [CL92].

- The technique of Landau and Vishkin [LV88] requires $\mathcal{O}(m)$ space. It preprocesses the input string in $\mathcal{O}(k^2 \cdot n)$ time and the pattern in $\mathcal{O}(m)$ time and uses Harel and Tarjan's [HT84] constant time algorithm for computing the lowest common ancestor (LCA for short) of two nodes of $cst(p\$)$.
- The method of Galil and Giancarlo [GG88] precomputes the compact suffix tree of $p\$$ and a summary of exact matches between p and t in $\mathcal{O}(m+n)$ space and time. Such a summary was later called matching statistics by Chang and Lawler [CL94].
- A second technique of Landau and Vishkin [LV89] takes $\mathcal{O}(k \cdot n)$ time and $\mathcal{O}(m+n)$ space. It preprocesses the pattern and the input string into a compact suffix tree and then applies an LCA algorithm.
- Galil and Park [GP90] devised a preprocessing method which takes $\mathcal{O}(m^2)$ space and time. It uses similar triples as the first technique of Landau and Vishkin, but does without a suffix tree and an LCA algorithm.
- A method of Ukkonen and Wood [UW93] requires $\mathcal{O}(m^2 \cdot |\mathcal{A}|)$ time using a simple form of the Aho-Corasick pattern matching machine [AC75]. A modification of the suffix automata construction in [Cro88] gives an $\mathcal{O}(m^2 + |\mathcal{A}|)$ time preprocessing method.
- The preprocessing technique of Chang and Lawler [CL94] takes $\mathcal{O}(m)$ space. It applies the less complicated constant time LCA algorithm of Schieber and Vishkin [SV88] and a new method to compute the matching statistics in $\mathcal{O}(m+n)$ time using only $cst(p\$)$.

In the following, we consider Diagonal Transition Algorithms in detail. In particular, section 6.6.1 explains their basic idea. In section 6.6.2, we describe the brute force Diagonal Transition Algorithm. Section 6.6.3 is devoted to preprocessing. We outline the method of Chang and Lawler and describe how the matching statistics can be used for the technique of Ukkonen and Wood. Finally, section 6.6.4 shows a functional implementation of the discussed techniques.

6.6.1 The Basic Idea

The monotonicity property of the D-diagonals suggests a more compact representation of table D. For each D-diagonal it suffices to store only the positions where the values of a D-diagonal increase.

Definition 6.6.2 For each $l, 0 \leq l \leq m$ and each $h, -l \leq h \leq n$ we define

$$C(l,h) = \max\{h+i \mid 0 \leq i \leq m, 0 \leq h+i \leq n, D(i,h+i) \leq l\}. \quad \square$$

The entries with value $\leq l$ on D-diagonal h end at column $C(l,h)$. Moreover, $C(l,h) - h$ is the row number of the last entry on D-diagonal h whose value is $\leq l$. Notice that $C(l,h)$ is welldefined for all $l, h, 0 \leq l \leq m$, $-l \leq h \leq n$. It is easily verified that for all D-diagonals $h, -m \leq h \leq n-m$ the following equivalence holds:

$$D(m, h+m) \leq k \iff C(k,h) = h+m. \tag{6.3}$$

Figure 6.4: Table C for $k = 2$, $p = adbbc$, and $t = abbdadcbc$

		h								
	C	-2	-1	0	1	2	3	4	5	6
	0			1	1	2	3	6	5	6
l	1		3	3	2	4	6	9	8	
	2	3	4	4	4	7	8	9		

Hence, table C can be used to solve the k-differences problem. If $D(m, j) \leq k$, then $m-k \leq j \leq n$. Since $D(m, m-k)$ is a value on D-diagonal $-k$ and $D(m, n)$ is a value on D-diagonal $n-m$, it suffices to compute $C(l, h)$ for all $l, h, 0 \leq l \leq k$, $-l \leq h \leq n-m+k-l$.

Example 6.6.3 [GP90] Let $k = 2$, $p = adbbc$, and $t = abbdadcbc$ as in Example 6.3.2. Figure 6.4 shows the entries $C(l, h)$, $0 \leq l \leq k$, $-l \leq h \leq n-m+k-l$ of table C.[5] Note that the shape of the entries form a parallelogram. For $h \in \{-2, -1, 2, 3, 4\}$ we obtain $C(k, h) = h + m$. Hence, the solutions to the k-differences problem are 3, 4, 7, 8, and 9. □

The definition of table C is consistent with [CL92]. However, it differs from [LV88, GG88, LV89, GP90, UW93, Ste94] where $C(l, h)$ is defined as the largest column $h + i$ such that $D(i, h + i) = l$. In most cases, this difference does not matter since the following property holds:

$$\text{If } h + i = C(l, h) \text{ and } h + i < lastcol(h) \text{ then } D(i, h + i) = l. \tag{6.4}$$

However, the condition $D(i, h + i) = l$ sometimes leads to inconsistencies as shown in the following example.

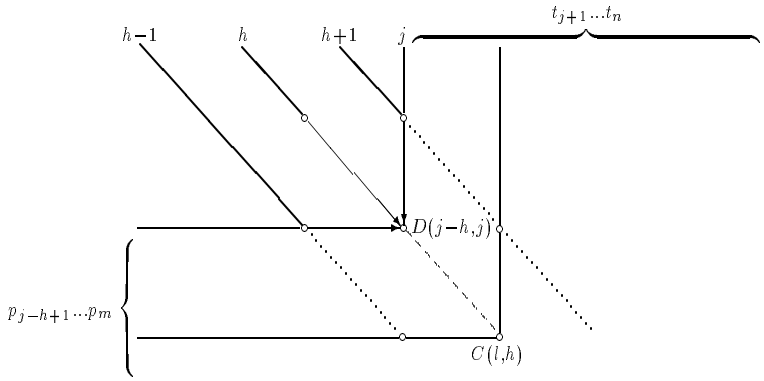
Example 6.6.4 Consider the D-diagonal 4 in Figure 6.3. It consists of the values 0, 0, 0, 1, 1, 1. There is no value 2 but a value 1 on this D-diagonal. According to Definition 6.6.2, we get $C(2, 4) = 9$ as shown in Figure 6.4. According to the definition in [LV88, GG88, LV89, GP90, UW93, Ste94], $C(2, 4)$ would be undefined. Galil and Park [GP90, Example 3] recognize this. But instead of correcting their definition of table C accordingly, they state that "$C(2, 4)$ is set to 9, the last column of D-diagonal 4." □

A method for computing table C first appeared in [LV88]. Later authors [GG88, LV89, GP90, UW93] used the method in almost identical form and only modified the computation of jumps. The idea of the method can be explained by Figure 6.5. Let us assume we want to compute $C(l, h)$ and already have obtained $C(l-1, h+1)$, $C(l-1, h)$, and $C(l-1, h-1)$. This means that in table D the entries of value $\leq l-1$ reach

- column $C(l-1, h+1)$ on D-diagonal $h+1$,
- column $C(l-1, h)$ on D-diagonal h,
- column $C(l-1, h-1)$ on D-diagonal $h-1$.

[5] Note that the corresponding table in the paper of Galil and Park [GP90, Table 3] contains several additional values which are needed for the boundary cases. We handle the boundary cases by explicit case distinction and therefore do not need the additional values.

Figure 6.5: Computation of $C(l,h)$ by the Diagonal Transition Algorithm



Let $j = \max\{C(l-1,h+1), C(l-1,h)+1, C(l-1,h-1)+1\}$. $D(j-h,j)$ gets a value $\leq l$ from one of the last entries of value $\leq l-1$ on the D-diagonals $h+1$, h, and $h-1$. These entries are found on the thick lines in Figure 6.5. As the next values on D-diagonal $h-1$m and $h+1$ on the dotted lines are not smaller than $D(j-h,j)$, we can apply Lemma 6.3.4, Statement 1. Hence, the values on D-diagonal h on the dashed line remain equal to $D(j-h,j)$ until there is mismatch between a pattern character in $p_{j-h+1}\ldots p_m$ and the corresponding input character in $t_{j+1}\ldots t_n$. Once a mismatch has been detected, the column $C(l,h)$ is reached.

Example 6.6.5 [GP90] Consider the computation of $C(2,2)$ in Example 6.6.3. We obtain column $j = \max\{C(1,3), C(1,2)+1, C(1,1)+1\} = \max\{6, 4+1, 2+1\} = 6$. So $D(6-2,6) = D(4,6)$ gets value 2 from $D(3,6) = 1$. There is a match $p_5 = t_7$ at column 7 of D-diagonal 2 which is the last match. Therefore, $C(2,2) = j+1 = 7$. □

So the basic idea to compute table C is relatively simple. However, we feel that the program for computing table C, as it is given in [LV88, GG88, LV89, GP90, UW93], is not easy to understand. This is for the following reasons:

- The program does not exactly compute the table as it was defined in the corresponding papers. This is due to the inconsistencies mentioned above (see Example 6.6.4).
- The program computes several fictitious table entries to handle the boundary cases which occur when either $C(l-1,h+1)$, $C(l-1,h)$, or $C(l-1,h-1)$ is undefined.

In our opinion, the computation of table C can easier be understood if one handles the boundary cases more explicitly by case distinction. We have done this in Lemma 6.6.6.

Lemma 6.6.6 Let $0 \leq l \leq k \leq m$ and $-l \leq h \leq n-m+k-l$. Suppose

$$j = \begin{cases} h, & \text{if } l = 0 \\ C(l-1,h+1), & \text{else if } h = -l \\ \max\{C(l-1,h+1), C(l-1,h)+1\}, & \text{else if } h = -l+1 \\ \max\{C(l-1,h+1), C(l-1,h)+1, C(l-1,h-1)+1\}, & \text{else if } h \geq -l+2 \end{cases}$$

Then $C(l,h) = \begin{cases} lastcol(h), & \text{if } j \geq lastcol(h) \\ j + jump(p_{j-h+1} \ldots p_m, t_{j+1} \ldots t_n), & \text{otherwise} \end{cases}$

Proof We first consider the case $l = 0$. By assumption $j = h < lastcol(h)$. It suffices to show that $C(0,h) = h + jump(p, t_{h+1} \ldots t_n)$. If $C(0,h) = h$, then $p_1 \neq t_{h+1}$. Hence, $C(0,h) = h + 0 = h + jump(p, t_{h+1} \ldots t_n)$. Let $C(0,h) \geq h + i + 1$ for some $i \geq 0$. Then $D(i, h+i) = 0$. Moreover, $D(i, h+i) \leq D(i, h+i+1)$ and $D(i, h+i) \leq D(i+1, h+i)$. By Lemma 6.3.4, Statement 1, $D(i, h+i) = D(i+1, h+i+1)$ if and only if $p_{i+1} = t_{h+i+1}$. Therefore, $C(0,h) = h + jump(p, t_{h+1} \ldots t_n)$.

Now suppose $l > 0$. It is easily verified that the following properties hold:

- $C(l-1, h+1)$ is defined.
- $C(l-1, h)$ is defined if $h \geq -l + 1$.
- $C(l-1, h-1)$ is defined if $h \geq -l + 2$.

Hence, j is defined. Consider the following cases:

1. If $j \geq lastcol(h)$, then $D(lastcol(h) - h, lastcol(h)) \leq l$ and hence $C(l,h) = lastcol(h)$.
2. Suppose $j < lastcol(h)$ and let $j' = j + jump(p_{j-h+1} \ldots p_m, t_{j+1} \ldots t_n)$. Obviously, $j' \leq lastcol(h)$ and $D(j' - h, j') = D(j - h, j) \leq l$ by Lemma 6.3.4, Statement 1. Assume that $D(j' - h, j') < l$. Then $C(l-1, h)$ exists and we obtain $C(l-1, h) \geq j \geq C(l-1, h) + 1$. This is a contradiction. Hence, $D(j' - h, j') = l$ and therefore $C(l,h) = j'$. □

The value j, as defined in Lemma 6.6.6, is very important for determining $C(l,h)$. We will call it *start column* for $C(l,h)$. From Lemma 6.6.6 it is not difficult to derive an algorithm that computes table C. This will be shown in section 6.6.4.

6.6.2 The Brute Force Diagonal Transition Algorithm

Definition 6.6.7 [GP90] Let $0 \leq cd \leq n - m + k$. C-diagonal cd, denoted by $Cdiag(cd)$, consists of the values $C(l,h)$, $0 \leq l \leq k$, $-l \leq h \leq n - m + k - l$ such that $l + h = cd$. Let $Cdiag(cd)(l) = C(l,h)$ be the l-th value of C-diagonal cd. □

Example 6.6.8 The C-diagonals in Figure 6.4 are as follows:

$$Cdiag(0) = (1,3,3), Cdiag(1) = (1,3,4), \ldots, Cdiag(6) = (6,8,9). \quad \square$$

The simplest Diagonal Transition Algorithm is Algorithm DTbf. It computes jumps in a brute force manner, and can be viewed as a variation of Ukkonen's [Ukk85a] or Myers' [Mye86] algorithm for calculating the unit edit distance. Algorithm DTbf corresponds to algorithm MN2 in [GP90].

Algorithm DTbf For each $cd, 0 \leq cd \leq n-m-k$ construct $Cdiag(cd)$ according to Lemma 6.6.6. Compute jumps brute force by pairwise character comparisons. Output $cd - k + m$ if $Cdiag(cd)(k) = cd - k + m$. □

Theorem 6.6.9 Algorithm DTbf correctly solves the k-differences problem in $\mathcal{O}(m)$ space and $\mathcal{O}(m \cdot n)$ worst case running time. The expected running time is $\mathcal{O}(k \cdot n)$.

Proof Let $h = cd - k$. By definition, $Cdiag(cd)(k) = cd - k + m$ if and only if $C(k, h) = h+m$. The correctness of the algorithm now follows from equivalence (6.3). Each C-diagonal contains $\mathcal{O}(k)$ entries. Since DTbf needs to store only three C-diagonals at any time, $\mathcal{O}(k)$ space suffices for the C-diagonals. The computation can be arranged such that the access to t is restricted to a sliding window of size $\mathcal{O}(m)$. Hence, the space requirement is $\mathcal{O}(m)$. There are $\mathcal{O}(n)$ C-diagonals to be computed. It takes $\mathcal{O}(k + cc)$ time to compute a C-diagonal where cc is the number of character comparisons to determine the jumps. In the worst case, $cc \in \mathcal{O}(m)$ since each comparison of p_i with t_{i+cd}, $1 \leq i \leq m$, can occur (see [GP90, Lemma 2]). Myers [Mye95b] showed that in the expected case $cc \in \mathcal{O}(k)$. This completes the proof. □

6.6.3 Efficient Computation of Jumps

The crucial point in the Diagonal Transition Algorithms is the calculation of jumps. To obtain a value $C(l, h)$, every Diagonal Transition Algorithm calculates the jump for some suffix of p and some suffix of t. In order to accomplish this calculation in constant time, one can preprocess p and t into the matching statistics.

Definition 6.6.10 Let $T = cst(p\$)$. For each suffix y of t $maxpsubword(y)$ is the longest prefix of y that is a subword of p. The *matching statistics* of t w.r.t. p is the function $mstats : \{y \mid t \sqsupset y, y \neq \varepsilon\} \to locations(T)$ defined by $mstats(y) = loc_T(maxpsubword(y))$. □

The term matching statistics is due to Chang and Lawler [CL94]. A similar notion (called "Best-Fit") already occurred in [GG88]. Note that the matching statistics in [CL94] consists of two tables $M(t, p)$ and $M'(t, p)$. These can easily be obtained from the function $mstats$. In particular,

$$M(t,p)(j) = |mstats(t_j \ldots t_n)|$$
$$M'(t,p)(j) = ceiling(mstats(t_j \ldots t_n))$$

for each $j, 1 \leq j \leq n$. Therefore, we will refer to $mstats$ when we explain how Chang and Lawler compute jumps.

In [CL94], a linear time algorithm for computing the matching statistics is devised. It traverses $cst(p\$)$ in a single left-to-right scan of the input string. The algorithm is presented as an imperative program that manipulates several pointers in a crucial way. Although invariants are given, we feel that the program is not easy to understand. Stephens [Ste94] gives a slightly improved presentation of Chang and Lawler' program, including an extensive example. However, the description is still imperative. We have devised a declarative and simpler description of the algorithm on the basis of locations. The two inner loops in Chang and Lawler's program roughly correspond to the functions *linkloc* and *scanprefix*, as defined in section 3.2.2.

Algorithm MS [CL94] Construct $T = cst(p\$)$ with all inner suffix links. Set $(loc_1, s_1) = scanprefix(loc_T(\varepsilon), t)$. For each $j, 1 \leq j \leq n-1$ successively compute

$$(loc_{j+1}, s_{j+1}) = \begin{cases} (linkloc(loc_j), \varepsilon), & \text{if } s_j = \varepsilon \\ scanprefix(loc_j, drop(1, s_j)), & \text{else if } loc_j \text{ is the } root \\ scanprefix(linkloc(loc_j), s_j), & \text{otherwise} \end{cases}$$

The following two lemmas show the correctness and the linearity of Algorithm MS.

Lemma 6.6.11 Algorithm MS computes for each $j, 1 \leq j \leq n$ a pair (loc_j, s_j) such that

1. $loc_j = mstats(t_j \dots t_n)$
2. $maxpsubword(t_j \dots t_n)s_j = t_j \dots t_n$.

Proof By induction on j. Let $j = 1$. Then $(loc_j, s_j) = scanprefix(loc_T(\varepsilon), t) = (loc_T(u), v)$ where $uv = t$ and u is the longest prefix of t such that $u \in words(T)$ (see Definition 3.2.9). Hence, $u = maxpsubword(t)$ which implies properties 1 and 2. Now suppose 1 and 2 hold for some $j, 1 \leq j \leq n-1$ and (loc_{j+1}, s_{j+1}) is determined according to Algorithm MS. Let $b = t_j$ and $y = t_{j+1} \dots t_n$. By case distinction, we show that $loc_{j+1} = mstats(y)$ and $maxpsubword(y)s_{j+1} = y$ hold.

- If $s_j = \varepsilon$, then $maxpsubword(by) = by$. Therefore, $loc_j \neq root$ and we obtain $maxpsubword(y)s_{j+1} = y$ where $s_{j+1} = \varepsilon$. Moreover, we get $loc_j = mstats(by) = loc_T(maxpsubword(by)) = loc_T(by)$. Thus, we can conclude $loc_{j+1} = linkloc(loc_j) = linkloc(loc_T(by)) = loc_T(y) = loc_T(maxpsubword(y)) = mstats(y)$.

- If $s_j \neq \varepsilon$, and loc_j is the $root$, then $maxpsubword(by) = \varepsilon$ and therefore $s_j = by$. Thus, $(loc_{j+1}, s_{j+1}) = scanprefix(root, drop(1, s_j)) = (loc_T(u), v)$, where $uv = drop(1, s_j) = y$ and u is the longest prefix of y such that $u \in words(T)$. Hence, $u = maxpsubword(y)$ and we can conclude $loc_{j+1} = mstats(y)$ and $us_{j+1} = y$.

- If $s_j \neq \varepsilon$ and loc_j is not the $root$, then $maxpsubword(by) = bw$ for some string w. Hence, $ws_j = y$. Since w is a prefix of $maxpsubword(y)$, we obtain $(loc_{j+1}, s_{j+1}) = scanprefix(linkloc(loc_j), s_j) = scanprefix(loc_T(w), s_j) = (loc_T(u), v)$ where $uv = ws_j = y$ and u is the longest prefix of y such that $u \in words(T)$. Hence, $u = maxpsubword(y)$. Thus, we can conclude $loc_{j+1} = mstats(y)$ and $us_{j+1} = y$. □

Note that the above proof is the first detailed correctness proof of Chang and Lawler's algorithm. In section 6.6.4, we will see that it is straightforward to translate Algorithm MS into a functional implementation.

Lemma 6.6.12 Algorithm MS needs $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ time.

Proof First note that at each node the number of outgoing edges is bound by $|\mathcal{A}_p|$. The suffix tree including inner suffix links can be constructed in $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time. The total time used by *scanprefix* is proportional to the number of character comparisons this function performs. Each input character in t is compared at most once successfully against a pattern character. Thus, *scanprefix* carries out at most n successful character comparisons. Every

unsuccessful character comparison either occurs while inspecting the list of outgoing edges at a node, or it does not lead to a recursive call of *scanprefix*. Hence, *scanprefix* performs at most $|\mathcal{A}_p| \cdot n$ unsuccessful character comparisons and therefore needs $\mathcal{O}(|\mathcal{A}_p| \cdot n)$ time altogether. By an analogous argumentation, one can show the like for *linkloc*. □

As already stated in [CL94], Algorithm MS has very much in common with McCreight's suffix tree algorithm [McC76]. In fact, the latter can be modified to compute *mstats*: Apply McCreight's algorithm to $T = cst(p\$)$ and t. Do not perform any operation that changes T, but output each location of a "head", and each "tail" encountered. This gives the sequence of pairs (loc_j, s_j) computed by Algorithm MS. Our unified description of McCreight's algorithm and Algorithm MS allows to verify this agreement formally. It should not be too difficult to obtain Algorithm MS from the function *mcc* (see section 3.5.3) by some simplifying program transformation steps.

In addition to $T = cst(p\$)$ and the matching statistics, the method of Chang and Lawler requires to preprocess a table *Leaf* of size $\mathcal{O}(m)$ such that $Leaf\ (x) = \overline{x\$}$ for each non-empty suffix x of p. (Note that such a table is not mentioned in [CL94].) Table *Leaf* can be computed in linear time by a single traversal of $cst(p\$)$. The traversal successively computes $loc_T(x)$ for shorter and shorter suffixes x of p and sets $Leaf\ (x) = ceiling(loc_T(x))$. Note that without the sentinel character a non-empty suffix of p may not correspond to a leaf. Hence, a table corresponding to table *Leaf* does not exist if we take $cst(p)$ instead of $cst(p\$)$. In other words, Chang and Lawler's method for computing jumps requires a sentinel character.

Since only $\mathcal{O}(m)$ recent values of the matching statistics must be stored at any time, Chang and Lawler's method requires $\mathcal{O}(m)$ space. In order to compute the jump for a suffix x of p and a suffix y of t in constant time, the following identity is exploited:

$$jump(x, y) = \min\{|mstats(y)|, |z|\} \tag{6.5}$$

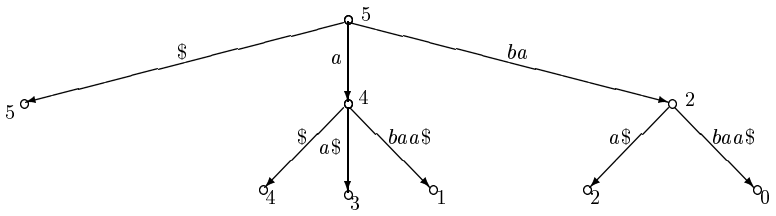
where $\overline{z}$ is the lowest common ancestor of the nodes $Leaf\ (x)$ and $ceiling(mstats(y))$ in T. To compute $\overline{z}$, the constant time LCA algorithm of Schieber and Vishkin [SV88] is used. Unfortunately, this algorithm is quite complicated and creates considerable overhead. Therefore, we feel that Chang and Lawler's method is in practice not the best solution to compute jumps in constant time. In our opinion, the following method of [UW93] is more practical.

Ukkonen and Wood [UW93] precompute three tables to obtain jumps:

- A table *maxprefix* equivalent to table $M(t, p)$ in [CL94] (see above).
- A table *witness* such that for each $j, 1 \leq j \leq n$ we have: $witness(j) = q$ implies that $q + 1$ is a position in p where $maxpsubword(t_j \ldots t_n)$ begins. Note that $witness(j)$ is not uniquely determined.
- A table *pjump* such that $pjump(x, x') = jump(x, x')$ for each pair x, x' of non-empty suffixes of p.

The table *maxprefix* and *witness* can be considered as another form of matching statistics. In [UW93], table *witness* is denoted by "M" and table *pjump* by "*Prefix*". *maxprefix* can be obtained from the function *mstats*. The same holds for the table *witness* if we annotate the suffix tree appropriately.

Figure 6.6: $cst(babaa\$)$ with the Annotation *suffixbegin*



Definition 6.6.13 Let $T = cst(p\$)$ and suppose that the edges outgoing from a node in T are ordered. The annotation *suffixbegin* of T is defined as follows:

$$suffixbegin(\overline{v}) = \begin{cases} m - |v| + 1, & \text{if } \overline{v} \text{ is a leaf} \\ suffixbegin(\overline{vcw}), & \text{otherwise} \end{cases}$$

where $\overline{v} \xrightarrow{cw} \overline{vcw}$ is the smallest edge outgoing from $\overline{v}$. □

The constraint "smallest edge" makes the annotation *suffixbegin* unambiguous and easily computable. If the edges outgoing from a node are represented by an ordered list, then a single bottom up traversal of $cst(p\$)$ in $\mathcal{O}(m)$ steps suffices to construct *suffixbegin*.

Example 6.6.14 Figure 6.6 shows $T = cst(babaa\$)$ with the annotation *suffixbegin*. The edges are ordered according to the first character of the edge label. □

If we apply Algorithm MS to $T = cst(p\$)$ with the annotation *suffixbegin*, then we obtain a function *mstats* such that $witness(j) = suffixbegin(ceiling(mstats(t_j \ldots t_n)))$ for each $j, 1 \leq j \leq n$. Hence, *mstats* subsumes both, Chang and Lawler's as well as Ukkonen and Wood's matching statistics.

Note that the computation of *maxprefix* and *witness* does not necessarily require the sentinel character. Suppose we have computed values loc_j by applying Algorithm MS to $cst(p)$. If we modify Definition 6.6.13 by deleting the term +1, then we get $maxprefix(j) = |loc_j|$ and $witness(j) = suffixbegin(ceiling(loc_j))$.

In [UW93], two devices are described to obtain table *maxprefix* and *witness* in $\mathcal{O}(n)$ steps. A simple form of the Aho-Corasick pattern matching machine [AC75] is constructed in $\mathcal{O}(m^2 \cdot |\mathcal{A}|)$ time and space. A modification of the suffix automaton (see [BBH+85, Cro88]) takes $\mathcal{O}(m^2 + |\mathcal{A}|)$ time and space to be constructed. We will instead use the compact suffix tree of $p\$$ and apply Algorithm MS.

Table *pjump* (see above) can be precomputed and stored in $\mathcal{O}(m^2)$ space. Each table entry can be obtained in constant time which is due to the following lemma.

Lemma 6.6.15 For all suffixes au and bv of p, the following recurrence holds:

$$pjump(au, bv) = \begin{cases} 1 + pjump(u, v), & \text{if } a = b \text{ and } u \neq \varepsilon \text{ and } v \neq \varepsilon \\ 1, & \text{else if } a = b \\ 0, & \text{otherwise} \end{cases}$$

Proof Obvious. □

In [LV88] and [GP90], an algorithm is given that fills table *pjump* diagonal by diagonal using $\mathcal{O}(m^2)$ space and time. It exploits the fact that *pjump* is symmetric which means that $pjump(x, x') = pjump(x', x)$ for each x, x'. An algorithm with the same complexity that proceeds row by row is described in section 6.6.4. Thus, the preprocessing for the method of Ukkonen and Wood takes $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n) + m^2)$ time and $\mathcal{O}(m^2)$ space altogether if one takes Algorithm MS to compute the matching statistics.

The following lemma corresponds to Lemma 2 in [UW93]. Using *mstats* and the annotation *suffixbegin*, it explains how Ukkonen and Wood compute jumps in constant time.

Lemma 6.6.16 [UW93] Let x be a suffix of p and y be a suffix t. Then

$$jump(x, y) = \min\{|mstats(y)|, pjump(x, p_{q+1} \ldots p_m)\},$$

where $q = suffixbegin(ceiling(mstats(y)))$.

Proof Let $u = maxpsubword(y)$. By definition, u is a prefix of $p_{q+1} \ldots p_m$ and therefore $p_{q+1} \ldots p_m = uz$ for some string z. Obviously, $jump(x, y) \leq |u|$ holds. If $|u| \leq jump(x, uz)$, then u is a prefix of x and y. Hence, $jump(x, y) \geq |u|$ which implies $jump(x, y) = |u| = \min\{|u|, jump(x, uz)\}$. If $|u| > jump(x, uz)$, then $jump(x, y) = jump(x, u) = jump(x, uz) = \min\{|u|, jump(x, uz)\}$. Thus, in both cases $jump(x, y) = \min\{|u|, jump(x, uz)\} = \min\{|mstats(y)|, pjump(x, p_{q+1} \ldots p_m)\}$. □

6.6.4 Implementation

In this section, we first show how to preprocess p and t in order to compute jumps according to Lemma 6.6.16. Moreover, we give a function that enumerates C-diagonals. The function abstracts from the different ways of computing jumps. Finally, we derive two implementations of Diagonal Transition Algorithms. An implementation of Algorithm DTbf and an implementation of an algorithm that uses matching statistics and table *pjump* to obtain jumps in constant time. The latter algorithm will be called DTpp in the sequel.

Matching Statistics

Besides the annotation *suffixbegin*, we need the annotation *depth* (see section 3.7.3) to determine the depth of a location. Both annotations are computed in constant time by the following annotation function.

```
depthsuffixbegin::num→annotationfunction α (num,num)
depthsuffixbegin m d es = (d,m-d+1), if es = []
                        = (d,i),     otherwise
                          where i = hd [i | (u,N es' link (d,i))←es]
```

Note that the second argument d of *depthsuffixbegin* is the depth of the actual node $\overline{v}$. If the ordered list es representing the edges outgoing from $\overline{v}$ is empty, then $\overline{v}$ is a leaf and $suffixbegin(\overline{v}) = m - |v| + 1 = m - h + 1$ according to Definition 6.6.13. If es is not empty, then $suffixbegin(\overline{v})$ is set to $suffixbegin(\overline{vcw})$ where $\overline{v} \xrightarrow{cw} \overline{vcw}$ is the smallest edge in es. Recall that the edges in es are ordered according to the first character of the label.

The matching statistics is represented by the list

$$mstatslist = [mstats(t_1 \ldots t_n), mstats(t_2 \ldots t_n), \ldots, mstats(t_n)].$$

The function *mstats* computes *mstatslist* according to Algorithm MS.

```
mstats::α→(string α)→(string α)→num→[location α (num,num)]
mstats sentinel t p m
  = mstats' (scanprefix loceps t)
    where loceps = LocN (cst (p++[sentinel],m+1) (depthsuffixbegin m))

mstats'::(location α β,string α)→[location α β]
mstats' (loc,s) = suffixlocs (const []) loc,                 if s = []
                = loc:mstats' (scanprefix loc (drop 1 s)),   if rootloc loc
                = loc:mstats' (scanprefix (linkloc loc) s), otherwise
```

At first, *mstats* constructs $T = cst(p\$)$ with the annotations *depth* and *suffixbegin*. Let $loceps = loc_T(\varepsilon)$. *mstats* calls the function *mstats'* with the argument (loc_1, s_1) according to Algorithm MS. *mstats'* closely resembles the case distinction of the algorithm. In the j-th call, *mstats'* has the argument (loc_j, s_j). If s_j is empty, then so is s_{j+1} and the rest of *mstatslist* consists of the locations of the non-empty suffixes of $t_j \ldots t_n$. These are easily obtained by applying the function (*suffixlocs* (*const* [ ])) to loc_j. If s_j is not empty, then *mstats'* returns a list with the first element loc_j. The rest of the list is constructed by applying *mstats'* to a pair (loc_{j+1}, s_{j+1}) which is determined according to the second and third case of Algorithm MS. Since the function *mstats* resembles Algorithm MS almost literally, it is clear that the implementation is optimal with a running time of $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ and a space consumption of $\mathcal{O}(m)$. Note that the functions *enumX* and *enumX'* (see section 4.5.1) are very similar to *mstats* and *mstats'*, respectively.

For each location *loc* in *mstatslist* the function *locdepthsuffixbegin* returns the pair $(|loc|, q)$ where $q = suffixbegin(ceiling(loc))$.

```
locdepthsuffixbegin::(location α (num,num))→(num,num)
locdepthsuffixbegin (LocN node) = seltag node
locdepthsuffixbegin (LocE node s (w,j) (N es link (h,q))) = (h-j,q)
```

To implement DTpp, we transform the matching statistics into a list *ppt* of preprocessed information for t. (Of course, *ppt* also depends on p.)

$$ppt \;=\; map\ locdepthsuffixbegin\ (mstats\ sentinel\ t\ p\ m).$$

Since each call of *locdepthsuffixbegin* takes constant time, *ppt* is constructed in $\mathcal{O}(m)$ space and $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ time.

Pjump

Let $pjumplist = [row_1, \ldots, row_m]$ such that

$$row_i = [pjump(p_i \ldots p_m, p_1 \ldots p_m), pjump(p_i \ldots p_m, p_2 \ldots p_m), \ldots, pjump(p_i \ldots p_m, p_m)]$$

for each $i, 1 \leq i \leq m$.

Example 6.6.17 For $p = babaa$ we get

$$\begin{array}{rl} pjumplist = [[5, & 0, \ 2, \ 0, \ 0], \\ & [0, \ 4, \ 0, \ 1, \ 1], \\ & [2, \ 0, \ 3, \ 0, \ 0], \\ & [0, \ 1, \ 0, \ 2, \ 1], \\ & [0, \ 1, \ 0, \ 1, \ 1]]. \end{array} \quad \square$$

The function *getpjumplist* computes *pjumplist* row by row starting with row_m. For each $i, 1 \leq i \leq m-1$, row_i is computed by evaluating $(nextrow\ p_i\ [row_{i+1}, \ldots, row_m])$.

```
getpjumplist::(string α)→[[num]]
getpjumplist p
  = foldr nextrow [] p
    where nextrow a [] = [map (equal a) p]
          nextrow a (row:rows)
            = foldr nextval [equal (last p) a] (zip2 p (drop 1 row)):row:rows
              where nextval (b,value) row' = 1+value:row', if a = b
                                           = 0:row',       otherwise
```

If $i = m$, then $[row_{i+1}, \ldots, row_m] = [\,]$. The j-th entry in row_m is 1 if $p_m = p_j$, and 0, otherwise. Hence, row_m is computed in $\mathcal{O}(m)$ steps by the expression $(map\ (equal\ p_m)\ p)$, where the function *equal* is defined as follows:

```
equal::α→α→num
equal a b = 1, if a = b
          = 0, otherwise
```

Let $1 \leq i \leq m-1$. Then row_i can be obtained from row_{i+1} in $\mathcal{O}(m)$ steps. The last entry in row_i is obviously computed by the expression $(equal\ (last\ p)\ p_i)$. The other entries in row_i depend on $p_1 \ldots p_{m-1}$ and the last $m-1$ entries of row_{i+1}. Let $b = p_j$ and $value = pjump(p_{i+1} \ldots p_m, p_{j+1} \ldots p_m)$ for some $j, 1 \leq j \leq m-1$. b and $value$ are obtained by zipping p and $(drop\ 1\ row)$. According to Lemma 6.6.15, we obtain

$$pjump(p_i \ldots p_m, p_j \ldots p_m) = \begin{cases} 1 + value, & \text{if } p_i = b \\ 0, & \text{otherwise} \end{cases}$$

Note that *getpjumplist* does not exploit the fact that *pjump* is symmetric.

For DTpp we need constant time access to each element in row_i for each $i, 1 \leq i \leq m$. Therefore, extension 1 is required to transform *pjumplist* into a list *ppp* of arrays which contain the preprocessed information for p.

$$ppp \ = \ map\ (makearray\ m)\ (getpjumplist\ p).$$

From the above it is clear that the computation of *ppp* takes $\mathcal{O}(m^2)$ time and space.

Computing C-Diagonals

For the implementation of the Diagonal Transition Algorithms the exact value of a table entry $C(l,h)$ does not matter if it is $\geq lastcol(h)$. Therefore, each entry $C(l,h)$ is represented by a value r such that $C(l,h) = \min\{lastcol(h), r\}$. We store r instead of $C(l,h)$ since this avoids a lot of minimum-operations and leads to a more efficient and simpler implementation. The correctness is not affected by this "inaccuracy", which can easily be checked against the proof of Lemma 6.6.6. However, we have to modify the condition for an approximate match. Since

$$C(k,h) = h+m \iff \min\{lastcol(h), r\} = h+m \iff r \geq h+m, \tag{6.6}$$

an approximate match occurs at position $h+m$ if and only if $r \geq h+m$.

Let j be the start column for $C(l,h)$ (see page 166). To compute a jump, DTbf needs access to the strings $p_{j-h+1} \ldots p_m$ and $t_{j+1} \ldots t_n$ (see Lemma 6.6.6). In the same situation, DTpp needs access to the pair

$$(depthofloc, q) = locdepthsuffixstart\ (mstats(t_{j+1} \ldots t_n)) \tag{6.7}$$

and a value $pjump(p_{j-h+1} \ldots p_m, p_{q+1} \ldots p_m)$ (see Lemma 6.6.16). Since t and p as well as the preprocessed counterparts *ppp* and *ppt* are represented by lists, such an access should not be accomplished globally, for instance, by dropping the first $j-h$ elements of p (*ppp*) and the first j elements of t (*ppt*). It is more efficient to hold the required information locally. Therefore, instead of using the column number h explicitly, we operate on a triple (j,x,y) such that

- $x = p_{j-h+1} \ldots p_m$ and $y = t_{j+1} \ldots t_n$ for DTbf.
- $x = ppp_{j-h+1} \ldots ppp_m$ and $y = ppt_{j+1} \ldots ppt_n$ for DTpp.

(j,x,y) is the *access triple* for j and h. Note that x and y are lazy lists. Hence, an access triple needs constant space. For better readability, we introduce a corresponding type synonym.

```
accesstriple α β == (num,[α],[β])
```

For DTbf an access triple is of type $(accesstriple\ \alpha\ \alpha)$ since x and y are both lists of type $[\alpha]$. For DTpp an access triple is of type $(accesstriple\ (array\ num)\ (num,num))$ since x is a list of arrays over type num and y is a list of num-pairs.

DTbf can simply apply the function *lcp* (see section 3.4.6) to x and y to obtain a jump. Now consider DTpp. If y is not empty, then the first element of y is the pair $(depthofloc, q)$ as defined in equation (6.7). If additionally x is not empty, then the first element of x is an array *pjumprow* such that $(lookup\ q\ pjumprow = pjump(p_{j-h+1} \ldots p_m, p_{q+1} \ldots p_m))$. Hence, the following function *jumppp* correctly computes the jump for $p_{j-h+1} \ldots p_m$ and $t_{j+1} \ldots t_n$ from x and y.

```
jumppp::[array num]→[(num,num)]→num
jumppp [] y = 0
jumppp x [] = 0
jumppp (pjumprow:xrest) ((depthofloc,q):yrest)
  = 0,                                    if depthofloc = 0
  = min2 depthofloc (lookup q pjumprow), otherwise
```

Obviously, *jumppp* takes constant time. It will be used as the *jump*-function for DTpp. For the rest of this section it does not matter whether x and y refer to strings or to lists of preprocessed information. The difference is simply handled by the *jump*-functions *lcp* and *jumppp*. In this way, we abstract from the different methods of computing jumps.

Let $cd \geq 0$ and suppose *startcols* is a list of access triples of the form (j_l, x_l, y_l) such that j_l is the start column of $Cdiag(cd)(l) = C(l, h)$. The function *newcdiag* computes $Cdiag(cd)$ from *startcols*. For each triple $(j, x, y) = (j_l, x_l, y_l)$ in *startcols*, the jump value *jumpval* of x and y is calculated by a function *jump*. This gives $C(l, h) = j + jumpval$. To obtain the corresponding access triple, *jumpval* elements are dropped from x and y.

```
newcdiag::([α]→[β]→num)→[accesstriple α β]→[accesstriple α β]
newcdiag jump startcols
  = map getcvalue startcols
    where getcvalue (j,x,y) = (j+jumpval,drop jumpval x,drop jumpval y)
                              where jumpval = jump x y
```

The running time of *getcvalue* is $\mathcal{O}(jumpval)$ since it takes $\mathcal{O}(jumpval)$ time to drop the first *jumpval* elements from x and y. This implies that *newcdiag* requires $\mathcal{O}(m)$ steps. Unfortunately, this also holds if *jumpval* is determined in constant time, which means that our implementation cannot achieve an $\mathcal{O}(k \cdot n)$ worst case running time.

Now it remains to show how to compute the start columns. According to Lemma 6.6.6, the start column j on D-diagonal h is obtained from some values referring to the *same* D-diagonal and the D-diagonals *above* and *below* it (see also Figure 6.5). To obtain the corresponding access triple for j and h, we introduce three functions that maintain the second and third component of an access triple properly.

```
above::(accesstriple α β)→(accesstriple α β)
above (col,x,y) = (col,drop 1 x,y)


same::(accesstriple α β)→(accesstriple α β)
same (col,x,y) = (col+1,drop 1 x,drop 1 y)


below::(accesstriple α β)→(accesstriple α β)
below (col,x,y) = (col+1,x,drop 1 y)
```

The correctness of these functions becomes clear if one considers Figure 6.5. If we get a start column from the D-diagonal *above*, we only drop the first element of x. If we get it from the *same* D-diagonal, we also drop the first element of y and increment the column number. If we get it from the D-diagonal *below*, we only drop the first element of y and increment the column number. Note that (*drop* 1) is the identity on the empty list (see Appendix B). Hence, the three functions are also defined for the boundary cases where either x or y is empty.

For convenience, we introduce a function *maxat* which computes the maximum of two access triples according to their first component. *maxat* takes constant time.

```
maxat::(accesstriple α β)→(accesstriple α β)→(accesstriple α β)
maxat (col,x,y) (col',x',y') = (col,x,y),    if col >= col'
                             = (col',x',y'), otherwise
```

Note that the predefined function $max2$ cannot be used instead of $maxat$. This is because $max2$ additionally compares x and x' if $col = col'$.

DTbf and DTpp use the same function $getCdiags$ to compute C-diagonals. In addition to the $jump$-function, $getCdiags$ has two arguments x and y. For DTbf x points to p and y points to t. For DTpp x points to ppp and y points to ppt.

```
getCdiags::([α]→[β]→num)→[α]→[β]→[[accesstriple α β]]
getCdiags jump x y
  = getCdiags' 2 (drop 2 y) cdiag0 cdiag1
    where getCdiags' cd [] cdiag cdiag' = []
          getCdiags' cd s cdiag cdiag'
            = cdiag:getCdiags' (1+cd) (drop 1 s) cdiag' cdiag''
              where cdiag'' = newcdiag jump ((cd,x,s):startcols)
                    startcols = [maxat (below e) (maxat (same e') (above e'')) |
                                 (e,e',e'')←zip3 cdiag cdiag' cdiag'']
          cdiag0 = newcdiag jump ((0,x,y):map above cdiag0)
          cdiag1 = newcdiag jump ((1,x,drop 1 y):[maxat (same e) (above e') |
                                                  (e,e')←zip2 cdiag0 cdiag1])
```

$getCdiags$ calls a function $getCdiags'$ with four arguments:

- the number cd of the previously computed C-diagonals. Note that $cd \geq 2$.
- the suffix $s = y_{cd+1} \ldots y_n$ of y.
- the last but one C-diagonal $cdiag = Cdiag(cd-2)$.
- the last C-diagonal $cdiag' = Cdiag(cd-1)$.

Note that the C-diagonals are represented by lists of access triples. If s is the empty list, then no C-diagonal is to be computed. Otherwise, $getCdiags'$ constructs a new C-diagonal $cdiag''$ and calls itself with "shifted" arguments. $cdiag''$ is computed by applying $newcdiag$ to the list $((cd, x, s) : startcols)$. Let $cd = l + h$ for some $l \geq 0$ and $h \geq -l$.

- If $l = 0$, then cd is the start column for $C(l,h)$ and $(cd, x_{cd-h+1}, y_{cd+1}) = (cd, x, s)$ is the access triple for cd and h.
- Suppose $l > 0$ and let (j_l, x_l, y_l) be an access triple for cd and h such that j_l is the start column for $C(l,h)$. (j_l, x_l, y_l) is obtained from the values e, e', and e'' which are generated from $cdiag$, $cdiag'$, and $cdiag''$, respectively. In particular, (j_l, x_l, y_l) is the maximum of $(below\ e)$, $(same\ e')$, and $(above\ e'')$.

Note that we use programming with unknowns to implement $getCdiags'$. $cdiag''$ depends on $startcols$, and $startcols$ depends on $cdiag''$. Initially, $getCdiags'$ is called with $cdiag0 =$

$Cdiag(0)$ and $cdiag1 = Cdiag(1)$. These values are computed in a similar way as $cdiag''$. However, $cdiag0$ only depends on itself, and $cdiag1$ only depends on $cdiag0$ and $cdiag1$.

Note that *getCdiags* is independent of k. It generates C-diagonals which are lazy lists of access triples. To solve the k-differences problem, only the first $k+1$ elements of these lists are evaluated. Since *newcdiag* takes $\mathcal{O}(m)$ steps, we obtain a running time of $\mathcal{O}(m \cdot n)$. Since only three C-diagonals are stored at any time, the space requirement for the C-diagonals is $\mathcal{O}(k)$.

DTbf and DTpp

The function *appDTbf* implements DTbf. *appDTbf* generates the list *cdiags* of C-diagonals. According to equivalence (6.6), $Cdiag(h+k)(k)$ must be compared to $r = h + m$, for each $h \leq -k$. Since h ranges over $[-k, -k+1, \ldots]$, r ranges over $[m-k, m-k+1, \ldots]$. Therefore, the list *cdiags* is zipped with the list $[m-k, ..]$ to obtain the proper values of r.

```
appDTbf::num→(string α)→(string α)→[num]
appDTbf k p t = [r | (r,cdiag)←zip2 [m-k..] cdiags; first (cdiag!k) >= r]
                where cdiags = getCdiags lcp p t
                      m = #p
```

From the above it is clear that *appDTbf* takes $\mathcal{O}(m \cdot n)$ time and $\mathcal{O}(m)$ space. Therefore, it is an optimal implementation of DTbf.

DTpp is implemented by the function *appDTpp* which differs from *appDTbf* as follows:

- It has an additional preprocessing phase to construct *ppp* and *ppt*.
- It computes jumps by the function *jumppp*.

Recall that DTpp requires extension 1.

```
appDTpp::α→num→(string α)→(string α)→[num]
appDTpp sentinel k p t
  = [r | (r,cdiag)←zip2 [m-k..] cdiags; first (cdiag!k) >= r]
    where cdiags = getCdiags jumppp ppp ppt
          m = #p
          ppp = map (makearray m) (getpjumplist p)
          ppt = map locdepthsuffixbegin (mstats sentinel t p m)
```

We have seen that *ppt* is constructed in $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ steps. The laziness guarantees that only $\mathcal{O}(m)$ values of *ppt* are stored at any time. The space and time requirement for the construction of *ppp* is $\mathcal{O}(m^2)$. As in *appDTbf*, the computation of a C-diagonal requires $\mathcal{O}(m)$ steps. Hence, the preprocessing effort does not pay off and we achieve a worst case running time of $\mathcal{O}(m \cdot n)$ which is not optimal. Note that an optimal implementation would store $\mathcal{O}(m)$ recent portions of *ppt* in an array. However, this requires complicated storage management which creates considerable overhead. We will not give such an implementation.

6.7 The Column Partition Algorithm

The basic idea of Chang and Lampe's Column Partition Algorithm [CL92] is to partition each column of table D into *runs* of consecutive natural numbers. For instance, the column $(0, 1, 1, 1, 2, 3)$ is partitioned into three runs $(0, 1)$, (1), and $(1, 2, 3)$. Formally, an entry $D(i, j)$ belongs to *run* l of column j if and only if $i - D(i, j) = l$. Obviously, if $D(i, j) + 1 = D(i+1, j)$, then both $D(i, j)$ and $D(i + 1, j)$ belong to the same run of column j. Moreover, we have $i - D(i, j) = i + 1 - (D(i, j) + 1) \leq i + 1 - D(i + 1, j)$ by Lemma 6.3.3, Statement 1. In other words, while i grows, $i - D(i, j)$ is non-decreasing.

In order to solve the k-differences problem, one determines where each run ends. More precisely, a table of end points of runs is computed.

Definition 6.7.1 For all $l \geq 0$ and all $j, 0 \leq j \leq n$ we define

$$\begin{aligned} &SP(0, j) = 0 \\ &SP(l + 1, j) = EP(l, j) + 1 \\ &EP(l, j) = \max\{i \mid 0 \leq i \leq m, i - D(i, j) \leq l\} \end{aligned}$$

$SP(l, j)$ is the *starting point* of run l in column j. $EP(l, j)$ is the *end point*. □

Table SP is introduced for ease of presentation. Neither table EP nor table SP explicitly occur in [CL92]. Recently, Stephens [Ste94] gave a presentation of the Column Partition Algorithm, including some additional examples. However, in contrast to our presentation, Stephens closely follows [CL92].

Note that a run can be empty. In such a case, $EP(l, j) < SP(l, j)$. Run 0 is not empty since $EP(0, j) \geq 0 = SP(0, j)$. In [CL92, Ste94], it is not clearly stated how the endpoints of runs are related to the solutions to the k-differences problem. The following lemma clarifies this relation.

Lemma 6.7.2 For all $j, 0 \leq j \leq n$, all $k < m$, and all $l \geq 0$ we have $D(m, j) \leq k$ if and only if $EP(m - k - 1, j) < m$.

Proof
$$\begin{aligned} D(m, j) \leq k &\iff m - D(m, j) \geq m - k \\ &\iff m - D(m, j) > m - k - 1 \\ &\iff \max\{i \mid 0 \leq i \leq m, i - D(i, j) \leq m - k - 1\} < m \\ &\iff EP(m - k - 1, j) < m. \quad \square \end{aligned}$$

Example 6.7.3 Let $k = 2$, $p = adbbc$, and $t = abbdadcbc$. Figure 6.7 shows the corresponding table of end points. Run 3 of column 9 is empty since $EP(3, 9) = 4 < SP(3, 9) = EP(2, 9) + 1 = 5$. The solutions to the k-differences problem are 3, 4, 7, 8, and 9, as can be easily read from row $m - k - 1 = 2$, according to Lemma 6.7.2. □

The following lemma states some basic properties of runs. The endpoints of runs are non-decreasing and no two consecutive runs may both be empty. These properties were already stated, but not proved in [CL92].

Figure 6.7: Table EP for $p = adbbc$, $t = abbdadcbc$, and each $l, 0 \leq l \leq 4$

		j									
	EP	0	1	2	3	4	5	6	7	8	9
	0	5	0	1	2	1	0	1	1	2	2
	1	5	5	2	2	3	4	1	2	2	3
l	2	5	5	5	3	4	5	5	4	3	4
	3	5	5	5	5	5	5	5	5	5	4
	4	5	5	5	5	5	5	5	5	5	5

Lemma 6.7.4 [CL92] Let $i = EP(l, j)$ for some $l \geq 0$ and some $j, 0 \leq j \leq n$. Then:

1. $EP(l + 1, j) \geq i$.
2. If $i < m$ and $EP(l + 1, j) = i$, then $EP(l + 2, j) > i$.

Proof

1. If $i = m$, then $EP(l+1, j) = i$. Suppose $i < m$. By assumption, $i+1-D(i+1, j) \geq l+1$. If $i + 1 - D(i + 1, j) = l + 1$, then $EP(l + 1, j) \geq i + 1 > i$. If $i + 1 - D(i + 1, j) > l + 1$, then $EP(l + 1, j) = i$.
2. By assumption, $i - D(i, j) \leq l$. Assume $EP(l+2, j) = i$. Then $i+1-D(i+1, j) > l+2$ which implies $D(i, j) \geq D(i + 1, j) + 2$. This is a contradiction to Lemma 6.3.3, Statement 1. Thus, $EP(l + 2, j) > i$. □

The following lemma gives a recurrence for computing table EP. The first part corresponds to Proposition 3, and the second part to Proposition 4 in [CL92]. While Chang and Lampe only give proof sketches, we spell out the complete proofs and additionally handle the boundary cases.

Lemma 6.7.5 [CL92] Let $i = EP(l, j)$ and $i' = SP(l, j)$ for some $l \geq 0$ and $j, 0 \leq j < n$. Then the following holds:

1. If $i = i' - 1$, then $EP(l, j + 1) = \min\{m, i'\}$.
2. If $i \geq i'$ and $H = \{h \mid i' \leq h \leq \min\{m - 1, i\}, p_{h+1} = t_{j+1}\}$, then

$$EP(l, j + 1) = \begin{cases} \min(H), & \text{if } H \neq \emptyset \\ i, & \text{if } H = \emptyset \text{ and } i = EP(l + 1, j) \\ i + 1, & \text{if } H = \emptyset \text{ and } i < EP(l + 1, j) \end{cases}$$

Proof

1. Let $i = i' - 1$. Then $l > 0$ and $i = EP(l-1, j)$. If $i = m$, then $EP(l-1, j) = m$ and therefore $EP(l, j+1) = m = \min\{m, i'\}$. Suppose $i < m$. Then $i - D(i, j) \leq l - 1$. Moreover, $EP(l+1, j) > i$ by Lemma 6.7.4, Statement 2. Thus, $i+1-D(i+1, j) \geq l+1$ which implies $D(i, j) = D(i+1, j) + 1$. Therefore, we get $D(i, j) \leq D(i+1, j+1) \leq D(i+1, j)+1 = D(i, j)$, that is, $D(i, j) = D(i+1, j+1)$. Thus, $i+1-D(i+1, j+1) = i+1-D(i, j) \leq l$. If $i+1 = m$, then trivially $EP(l, j+1) = m = \min\{m, i'\}$. If $i+1 < m$, then $D(i+2, j+1) \leq D(i+1, j) + 1 \leq D(i, j)$ and therefore $i + 2 - D(i+2, j+1) \geq i+2-D(i, j) = i+1-D(i+1, j) \geq l+1$ which implies $EP(l, j+1) = i+1 = \min\{m, i'\}$.

2. Let $i \geq i'$ and $H = \{h \mid i' \leq h \leq \min\{m-1, i\}, p_{h+1} = t_{j+1}\}$. We first show that $i' \leq EP(l, j+1) \leq i+1$. If $i' = 0$, then $i' \leq EP(l, j+1)$ obviously holds. If $i' > 0$, then $i'-1-D(i'-1, j) \leq l-1$ and $i'-D(i', j) \geq l$. Hence, $i'-D(i'-1, j) \leq i'-D(i', j)$ and therefore $D(i', j) \leq D(i'-1, j) \leq D(i', j+1)$. Thus, we can conclude $i'-D(i', j+1) \leq i' - D(i'-1, j) \leq l$ which implies $i' \leq EP(l, j+1)$. If $i+1 = m$, then we obviously get $EP(l, j+1) \leq m$. If $i+1 < m$, then $D(i+2, j+1) \leq D(i+1, j) + 1$. Moreover, $i - D(i, j) \leq l$ and $i + 1 - D(i+1, j) \geq l + 1$. Thus, $D(i+1, j) \leq D(i, j)$ and we can conclude $i + 2 - D(i+2, j+1) \geq i + 2 - (D(i+1, j) + 1) = i + 1 - D(i+1, j) \geq i + 1 - D(i, j) \geq i' + 1 - D(i', j) \geq l + 1$. This implies $EP(l, j+1) \leq i + 1$. Consider the following three cases:

   - If $H \neq \emptyset$, then let $h = \min(H)$. We have $p_{h+1} = t_{j+1}$ and $p_{h'+1} \neq t_{j+1}$ for all $i' \leq h' < h$. Hence, $D(h, j) = D(h+1, j+1)$ which implies $h+1-D(h+1, j+1) = h + 1 - D(h, j) = l + 1$. If $h = i'$, then $h - D(h, j+1) \leq l$ as shown above. Suppose $h > i'$. Then $D(h, j) \geq D(h-1, j)$ and $D(h-1, j+1) \geq D(h-1, j)$. Hence, $D(h, j+1) = D(h-1, j) + 1$ by Lemma 6.3.4, Statement 1. Thus, $h - D(h, j+1) = h - (D(h-1, j) + 1) = h - 1 - D(h-1, j) \leq h - D(h, j) = l$, that is, $EP(l, j+1) = h$.
   - Suppose $H = \emptyset$ and $i = EP(l+1, j)$. Then $D(i, j) \leq D(i, j+1)$ and $D(i+1, j)+1 = D(i, j)$. Therefore, $D(i+1, j+1) = D(i, j)$ and $i - D(i, j+1) \leq i - D(i, j) = l$. Moreover, $i+1-D(i+1, j+1) = i+1-D(i, j) = l+1$, that is, $EP(l, j+1) = i$.
   - Suppose $H = \emptyset$ and $i < EP(l+1, j)$. Then $D(i+1, j) = D(i, j)$ and $D(i, j) \leq D(i, j+1)$. Hence, $D(i, j) + 1 = D(i+1, j+1)$ since $p_{i+1} \neq t_{j+1}$. Thus, we can conclude $i+1-D(i+1, j+1) = i-D(i, j) = l$ which implies $EP(l, j+1) \geq i+1$. Since $EP(l, j+1) \leq i+1$, we get $EP(l, j+1) = i+1$. □

In order to obtain each entry of table EP in constant time, the input alphabet and the pattern is preprocessed. In particular, a function $position : \mathcal{A} \times \{0, \ldots, m\} \to \{0, \ldots, m\}$ is precomputed:

$$position(b, i') = \min(\{h \mid i' \leq h \leq m-1, b = p_{h+1}\} \cup \{m\})$$

With the assumptions of Lemma 6.7.5, the following holds:

$$H \neq \emptyset \iff position(t_{j+1}, i') \leq \min\{m-1, i\} \tag{6.8}$$

$$\text{If } H \neq \emptyset, \text{ then } \min(H) = position(t_{j+1}, i') \tag{6.9}$$

The function $position$ can be efficiently represented by a table which contains for each $b \in \mathcal{A}$ the list of positions in p where b occurs. Such a table can be precomputed in $\mathcal{O}(|\mathcal{A}| + m)$

space and time. However, it does not allow to evaluate $position(b, i')$ in constant time. An alternative solution is to directly represent the function *position* by an $|\mathcal{A}| \times (m+1)$-table. This can be precomputed in $\mathcal{O}(|\mathcal{A}| \cdot m)$ time according to the following equation:

$$position(b, i') = \begin{cases} i', & \text{if } i' = m \text{ or } b = p_{i'} \\ position(b, i'+1), & \text{otherwise} \end{cases} \tag{6.10}$$

We prefer the second representation since it allows to compute each entry of table *EP* in constant time. However, according to Chang and Lampe both representations lead to the same running time in practice.

To solve the k-differences problem, one computes table *EP* and outputs j if $EP(m-k-1, j)$ is smaller than m (see Lemma 6.7.2). Note that it does not suffice to evaluate the entries $EP(0, j), EP(1, j), \ldots, EP(m-k-1, j)$ of column j since $EP(l, j)$ may depend on the entry $EP(l+1, j-1)$ for some $l \geq 0$. In other words, it is not clear which entries of each column must be computed. Therefore, the evaluation of table *EP* requires a demand driven strategy. This can easily be realized in a lazy functional language (see section 6.7.1).

A more efficient evaluation strategy computes only the essential entries of table *EP*, that is, the entries smaller than m. This strategy leads to the following algorithm which is called *kn.clp* in [CL92].

Algorithm CP [CL92] Construct an $|\mathcal{A}| \times (m+1)$-table representing function *position*. For each $j, 1 \leq j \leq n$ and each $l \geq 0$ compute $EP(l, j)$ according to Lemma 6.7.5 until $EP(l, j) = m$. Store only the essential entries. If a column contains at least $m - k$ essential entries, then output j. □

Theorem 6.7.6 Let q be the average of row m in table D. Algorithm CP correctly solves the k-differences problem in $\mathcal{O}(|\mathcal{A}| \cdot m + (m-q) \cdot n)$ time and $\mathcal{O}(|\mathcal{A}| \cdot m)$ space.

Proof The correctness follows from the previous lemmas. The preprocessing phase takes $\mathcal{O}(|\mathcal{A}| \cdot m)$ space and time. Let $l = \max\{l \mid l \geq 0, EP(l, j) < m\}$. Obviously, Algorithm CP evaluates $l+1$ entries in column j. Due to the preprocessing, each of the entries is evaluated in constant time. Now note that $EP(l, j) < m$ implies $m - D(m, j) > l$, and that $EP(l+1, j) \geq m$ implies $m - D(m, j) \leq l+1$. Thus, $l+1 = m - D(m, j)$, that is, the search phase takes $\mathcal{O}((m-q) \cdot n)$ time. Since at each time only two columns of table *EP* must be stored, the space requirement for the search phase is $\mathcal{O}(m)$. □

Since q can be zero, Algorithm CP takes $\mathcal{O}(m \cdot n)$ time in the worst case. According to [CL92], it is always faster than UKKco. Hence, the average case running time is $\mathcal{O}(k \cdot n)$. Note the dependence of the algorithm on the row average q: the larger q, the faster becomes Algorithm CP. q grows with the size of the input alphabet. In particular, the observed running time is $\mathcal{O}(k \cdot n / \sqrt{|\mathcal{A}|} - 1)$ (see [CL92, CM94]). According to Chang and Lampe, Algorithm CP was measured to be considerably faster than other algorithms. For instance, it improves on DTpp by a factor 2.5 for $|\mathcal{A}| = 2$, by a factor 4 for $|\mathcal{A}| = 4$, and by a factor 10 for $|\mathcal{A}| = 20$.

Before we consider implementation issues, we note that Chang and Marr [CM94] claim that the column partition technique can also be applied to the table E_δ when δ is the unit cost function. This may lead to an improved method for computing the unit edit distance. Unfortunately, Chang and Marr do not substantiate their claim.

6.7.1 Implementation

To implement the Column Partition Algorithm, extension 1 is required. We assume that the input alphabet $\mathcal{A}$ is given as a pair $(characters, encode)$ of type $(alphabet\ \alpha)$. The function *position* is represented by a two-dimensional array *posarray* such that for each $b \in \mathcal{A}$ and each $h, 0 \leq h \leq m$ we have $(lookup\ h\ (lookup\ (encode\ b)\ posarray) = position(b, h))$. *posarray* is computed in $\mathcal{O}(|\mathcal{A}| \cdot m)$ time by the function *getposarray*. For each $b \in \mathcal{A}$, *getposarray* calls the function $(getpos\ b)$ to obtain the list $[position(b, 0), position(b, 1), \ldots, position(b, m)]$ from right to left according to equation (6.10). This list is transformed into an array of length $m + 1$ using the function *makearray*. Finally, the resulting list of arrays is transformed into an array of arrays of length $|\mathcal{A}|$.

```
getposarray::(alphabet α)→(string α)→num→array (array num)
getposarray (characters,encode) p m
  = makearray l (map (makearray (m+1).getpos) characters)
    where l = #characters
          getpos b = foldr prev [m] (zip2 [0..] p)
                     where prev (i',a) (i:is) = i':i:is,  if b = a
                                              = i:i:is,   otherwise
```

Note that Stephens independently gave an imperative program that is (essentially) equivalent to the function *getposarry* (see [Ste94, Figure 5.17]).

In the following, we first define a function *colpart* that completely computes each column of table *EP*. From *colpart* we develop in a systematic way a variation *colparte* that computes only the essential entries.

For each $j, 0 \leq j \leq n$ column j of table *EP* is represented by the list $endpoints_j = [EP(0, j), EP(1, j), EP(2, j), \ldots]$. Let $(posarrayb = lookup\ (encode\ b)\ posarray)$ where $b = t_{j+1}$. Then the expression $(colpart\ m\ endpoints_j\ posarrayb)$ returns the list $endpoints_{j+1}$. Each entry in this list is calculated in constant time by the function *nextep*. In particular, for each $l \geq 0$, *nextep* is called with the arguments $i' = SP(l, j)$ and $(i : eps) = [EP(l, j), EP(l + 1, j), EP(l + 2, j), \ldots]$ to return the list $[EP(l, j + 1), EP(l + 1, j + 1), EP(l + 2, j + 1), \ldots]$. The entry $epl = EP(l, j + 1)$ is calculated according to the case distinction in Lemma 6.7.5. To decide $H = \emptyset$ and to extract $\min(H)$, properties 6.8 and 6.9 are exploited after evaluating $h = position(b, i') = position(t_{j+1}, i')$ by a lookup in array *posarrayb*.

```
colpart::num→[num]→(array num)→[num]
colpart m endpoints posarrayb
  = nextep 0 endpoints
    where nextep i' (i:eps)
            = epl:nextep (i+1) eps
              where h = lookup i' posarrayb
                    epl = min2 m i', if i+1 = i'
                        = h,         if h <= min2 (m-1) i
                        = i,         if i = hd eps
                        = i+1,       otherwise
```

To solve the k-differences problem, one begins with column $endpoints_0 = [m, m, \ldots]$ and computes the list $[endpoints_1, \ldots, endpoints_n]$ using the function *colpart*. If the entry

$endpoints_j(m-k-1)$ is smaller than m, then one outputs j (cf. Lemma 6.7.2). Note that each list $endpoints_j$ is infinite. However, the laziness guarantees that only those entries are evaluated which are required to determine the value $endpoints_{j'}(m-k-1)$ for some $j' \geq j$.

For the implementation of Algorithm CP each column of table *EP* is represented by a list of its essential entries. More precisely, column j is represented by the list $eendpoints_j = [EP(0,j), EP(1,j), \ldots, EP(l,j)]$, where l is the maximal index such that $EP(l,j) < m$. This implies that the first column of table *EP* is represented by the empty list.

The most important part of the implementation is a function *nexteep* that computes only the essential entries of an *EP*-column. In other words, *nexteep* transforms the list $eendpoints_j$ into the list $eendpoints_{j+1}$. We show how to develop *nexteep* from *nextep* in a systematic way by some program transformation steps. At first we copy the definition of function *nextep* and replace every occurrence of the identifier *nextep* by *nexteep*. With the resulting definition we proceed as follows:

- Suppose *nexteep* is called with the arguments i' and $(i : eps)$. Let $eps \neq [\,]$. Recall that $i' = SP(l,j)$, $i = EP(l,j)$, and $(hd\ eps = EP(l+1,j))$ for some $l \geq 0$. Since i and $(hd\ eps)$ occur in $eendpoints_j$, we conclude $i \leq hd\ eps < m$. Under these conditions, *nexteep* is simplified which leads to the following definition for the value *epl*:

```
epl = i',    if i+1 = i'
    = h,     if h <= i
    = i,     if i = hd eps
    = i+1,   otherwise
```

  If $i+1 \neq i'$, then $epl < m$. Suppose $i+1 = i'$. Then $h > i$ and $(i < hd\ eps)$. Hence, $i' \leq hd\ eps < m$ which implies $epl < m$. Moreover, the fourth equation defining *epl* subsumes the first equation which allows a further simplification. The result is shown in 1 below.

- The equation for *nexteep* obtained in the previous step is copied and transformed under the condition that $i < m$ and $eps = [\,]$ hold. This leads to a definition for the case where the second argument of *nexteep* is the list $[i]$. See 2 below.

- The equation obtained in the previous step is copied and transformed under the condition that $i = m$ holds. This leads to a definition for the case where the second argument of *nexteep* is the empty list. See 3 below.

Altogether, we end up with the following modification of *colpart*.

```
colparte::num→[num]→(array num)→[num]
colparte m eendpoints posarrayb
  = nexteep 0 eendpoints
    where nexteep i' [] = [h], if h < m                          || 3
                        = [],  otherwise
                          where h = lookup i' posarrayb
          nexteep i' [i] = h:nexteep (i+1) [],    if h <= i      || 2
                         = i+1:nexteep (i+1) [],  if i+1 < m
                         = [],                    otherwise
                           where h = lookup i' posarrayb
          nexteep i' (i:eps) = epl:nexteep (i+1) eps              || 1
                             where h = lookup i' posarrayb
                                   epl = h,   if h <= i
                                       = i,   if i = hd eps
                                       = i+1, otherwise
```

It is easily verified that *nexteep* only returns essential entries. Moreover, our systematic development guarantees that *all* essential entries are computed by *nexteep*. Therefore, the definition above is correct.

Algorithm CP is implemented by the function *appCP*. It preprocesses the pattern and the input alphabet into the two-dimensional array *posarray*. This takes $\mathcal{O}(|\mathcal{A}| \cdot m)$ time and space. From this a list *posarrayt* is constructed which contains for each $b \in \{t_1, t_2, \ldots, t_n\}$ a reference to the array *posarrayb* as defined above. The initial column of table *EP* is represented by the empty list. Successively applying the function (*colparte* m) yields the list of columns containing the essential entries of table *EP*. An approximate match occurs if and only if there are at least $m - k$ interesting entries in a column. To test the latter condition, the first $m - k - 1$ elements of a list representing a column are dropped. If the remaining list is non-empty, then $EP(m - k - 1, j) < m$. This technique avoids unnecessary evaluations of column entries.

```
appCP::(alphabet α)→num→(string α)→(string α)→[num]
appCP (characters,encode) k p t
  = [j | (j,eendpoints)←zip2 [0..] (scanl (colparte m) [] posarrayt);
         drop (m-k-1) eendpoints ~= []]
    where m = #p
          posarray = getposarray (characters,encode) p m
          posarrayt = [lookup (encode b) posarray | b←t]
```

The preprocessing phase in *appCP* takes $\mathcal{O}(|\mathcal{A}| \cdot m)$ time and space. It is easily verified that *nexteep* computes each column entry in constant time. Thus, each column is obtained in $\mathcal{O}(m - q)$ steps where q is the average value of row m in table D. Therefore, *appCP* is an optimal implementation of CP.

6.8 Chang and Lawler's Filtering Technique

For a random string t the number of solutions to the k-differences problem is very small compared to n. The running time of an algorithm that solves the k-differences problem is therefore dominated by the effort needed to verify for almost all positions that there is no approximate match. This means that a fast algorithm should identify approximate mismatches fast. Chang and Lawler's linear and sublinear expected time algorithms [CL94] do so by applying a filter to t. The algorithms discard a subword of t if it can be inferred that it does not contain an approximate match. The remaining subwords are marked as interesting. To each interesting subword a dynamic programming algorithm is applied to verify if there is indeed an approximate match.

In this section, we give a declarative presentation of Chang and Lawler's filtering technique. Since it is based on maximal matches (note that this is not mentioned in [CL94]), we use the notion of partitions. In this way, we obtain a very concise and intuitive description, and can give simplified correctness proofs which argue about subwords, submatches, and marked characters, rather than complicated index expressions.

Some of the ideas and proofs presented in this section were developed in joint work with Enno Ohlebusch.

6.8.1 Linear Expected Time Algorithm

The linear expected time algorithm of Chang and Lawler (LET for short) is based on the following two observations:

- An approximate match is at least of length $m - k$.
- An approximate match contains at most $k + 1$ marked characters of the partition $\Psi_{lr}(t, p)$.

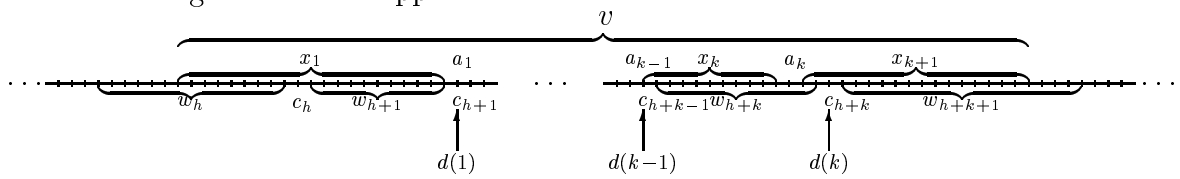
Thus, a subword of t which is shorter than $m - k$ and which contains $k + 1$ consecutive marked characters, can be discarded since it does not contain an approximate match. The remaining subwords may contain an approximate match. Hence, they are considered as interesting subwords. This is formally stated in the following algorithm.

Algorithm LET [CL94] Compute $\Psi_{lr}(t, p) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$. If $r \leq k$, then t is an interesting subword. Otherwise, for each $h, 1 \leq h \leq r - k$, let $s_h = w_h c_h \ldots w_{h+k} c_{h+k} w_{h+k+1}$. If $|s_h| \geq m-k$, then s_h is an interesting subword. Merge all overlapping interesting subwords. For each interesting subword obtained in such a way solve the k-differences problem. □

In an implementation of Algorithm LET, one represents the partition $\Psi_{lr}(t, p)$ by some pointers into the input string. In [CL94], the pointers refer to the first position of each subword $w_h c_h$. We found it convenient to have pointers to the marked characters instead:

Definition 6.8.1 Consider the partition $\Psi_{lr}(t, p) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$. For each $h, 1 \leq h \leq r$ let mp_h be the position of the marked character c_h in t. Additionally define $mp_0 = 0$ and $mp_{r+1} = n + 1$ as boundary values. □

Figure 6.8: An Approximate Match with Exact k Differences



Theorem 6.8.2 [CL94] Algorithm LET correctly solves the k-differences problem.

Proof Let $edist_\delta(p, v) \leq k$ for some subword v of t which begins at position j. An approximate match contained in an interesting subword will be detected by Algorithm LET. Hence, to prove the correctness, it suffices to show that v is a subword of an interesting subword. According to Lemma 3.9.13, there is an $l, 0 \leq l \leq k$ and a partition $\Psi = (x_1, a_1, \ldots, x_l, a_l, x_{l+1})$ of v w.r.t. p. Moreover, $mp_{h-1} < j \leq mp_h$ for some $h, 1 \leq h \leq r+1$. Consider the following cases:

- If $r \leq k$, then t is an interesting subword and v is a subword of t.
- If $r > k$ and $h > r - k$, then v is a subword of $w_h c_h \ldots w_r c_r w_{r+1}$ which is a subword of $s_{r-k} = w_{r-k} c_{r-k} \ldots w_r c_r w_{r+1}$. Hence, v is a subword of s_{r-k}. Since $|v| \geq m - k$, we conclude $|s_{r-k}| \geq m - k$. Thus, s_{r-k} is an interesting subword.
- If $r > k$ and $h \leq r - k$, then mp_{h+i} is defined for all $i, 0 \leq i \leq k+1$. One shows by induction on i that $mp_{h+i} \geq j + |x_1 a_1 \ldots x_i|$ holds for all $i, 0 \leq i \leq l+1$ (see also Theorem 6.8.5). This implies $mp_{h+k+1} \geq mp_{h+l+1} \geq j + |v|$. Hence, v is a subword of $s_h = w_h c_h \ldots w_{h+k} c_{h+k} w_{h+k+1}$. Since $|v| \geq m - k$, we have $|s_h| \geq m - k$. Thus, s_h is an interesting subword. □

Note that Theorem 6.8.2 precisely handles the boundary cases of Algorithm LET which occur when $r \leq k$ or $h > r - k$. These cases are not considered in [CL94].

The proof of Theorem 6.8.2 gives a precise analysis of the relation between the submatches and the marked characters in the partitions Ψ and $\Psi_{lr}(t, p)$. In particular, it shows that the prefix $x_1 a_1 \ldots x_i a_i$ of the approximate match v ends at least at the marked character c_{h+i}. Thus, mp_{h+i} is the maximal position in t where the i-th difference a_i in v is guaranteed to occur (if there is one). This is illustrated in Figure 6.8 which shows a possible arrangement of an approximate match $v = x_1 a_1 \ldots x_k a_k x_{k+1}$ with exactly k differences $a_1, \ldots, a_k$. $d(i)$ points to the maximal position in t where the i-th difference a_i can occur.

To verify whether an interesting subword indeed contains an approximate match, Chang and Lawler apply their version of the Diagonal Transition Algorithm (see section 6.6.3). One reason for this is the space efficiency and the good average and worst case running time of $\mathcal{O}(k \cdot n)$. Another reason is that the Diagonal Transition Algorithm uses the matching statistics of t w.r.t. p as a preprocessed information. This can also be used to compute the positions $mp_0, mp_1, \ldots, mp_r, mp_{r+1}$ of the marked characters in $\Psi_{lr}(t, p) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ without much extra effort. Let $T = cst(p\$)$ and $y = t_{mp_h+1} \ldots t_n$ for some $h, 0 \leq h \leq r$. Then according to Definitions 6.6.10 and 3.9.10 we have: $mp_{h+1} = mp_h + 1 + |w_{h+1}| = mp_h + 1 + |maxpsubword(y)| = mp_h + 1 + |loc_T(maxpsubword(y))| = mp_h + 1 + |mstats(y)|$.

Hence, mp_{h+1} can be obtained in constant time from *mstats*. Altogether, the preprocessing time of Algorithm LET is dominated by the time to compute the matching statistics. According to Lemma 6.6.12, this is $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$.

We emphasize that the Diagonal Transition Algorithm can be replaced by any other algorithm that solves the k-differences problem. For instance, if one takes Algorithm UKKco or Algorithm CP, then the full generality of the matching statistics is not needed and the positions of the marked characters can directly be obtained from $\Psi_{lr}(t,p)$. This is computed from $T = cst(p)$ in $\mathcal{O}(|\mathcal{A}_p| \cdot n)$ steps as described in section 3.9.2.

The different phases of Algorithm LET can be meshed. Thus, only $\mathcal{O}(k)$ marked positions mp_h and $\mathcal{O}(k \cdot m)$ input characters must be stored at any time. Of course, this requires extra buffering operations which can be accomplished in $\mathcal{O}(n)$ time altogether. Note that Chang and Lawler derive a space bound of $\mathcal{O}(m)$. It seems that they do not count the space for storing the input characters.

Let us call a position in t interesting if it is included in one of the interesting subwords. The overall running time of Algorithm LET is $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n) + ipos \cdot iwork)$ where *ipos* is the number of interesting positions in t and *iwork* is the average amount of work done for each interesting position. In the worst case, $ipos = n$ and $iwork = k$ ($iwork = m$ if we take Algorithm UKKco or Algorithm CP). For the expected case Chang and Lawler derive the following result:

Theorem 6.8.3 Let t be a uniformly random string and $k^* = m/(\log_{|\mathcal{A}|} m + c_1) - c_2$ where c_1 and c_2 are constants depending on $|\mathcal{A}|$. If $k < k^*$, then $ipos \cdot iwork \in \mathcal{O}(n)$.

Proof See [CL94]. □

Ukkonen [Ukk92a] independently devised a filtering algorithm that is based on maximal matches. A closer look reveals that it is virtually the same as Algorithm LET. Experimental results in [JTU91] show that Ukkonen's Algorithm is faster than other algorithms when m is large and k is not too large relative to m. This behavior is consistent with Theorem 6.8.3. We do not know an experimental comparison of Ukkonen's algorithm and Algorithm LET.

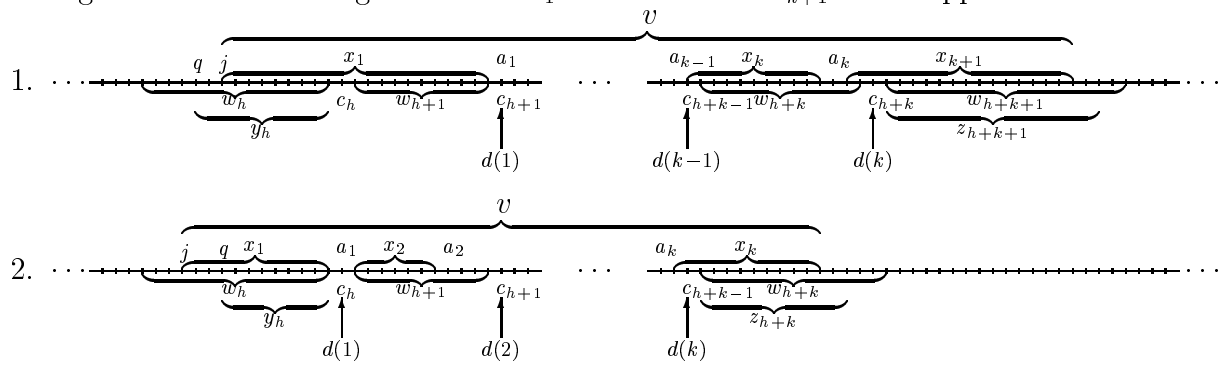
The algorithms described in the following three sections have a structure similar to Algorithm LET. They mainly differ in how they compute interesting subwords.

6.8.2 Improved Linear Expected Time Algorithm

In this section, we show how to improve the filter applied in Algorithm LET. The idea is to not totally ignore the arrangement of the subwords of p in an approximate match. In particular, we exploit the fact that an approximate match has a partition of size $l \leq k$ which begins with a prefix and ends with a suffix of p (see Lemma 3.9.13). This leads to a more sensitive filter which requires some additional computational effort. However, we will show that the preprocessing still takes $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ time. So the improved sensitivity may pay off.

Definition 6.8.4 Consider the partition $\Psi_{lr}(t,p) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$. Define $y_1 = w_1$ and $z_{r+1} = w_{r+1}$. For each $h, 2 \leq h \leq r+1$ let y_h be the longest suffix of w_h that is a prefix of p. For each $h, 1 \leq h \leq r$ let z_h be the longest prefix of w_h that is a suffix of p. □

Figure 6.9: Considering the Prefix x_1 and the Suffix x_{k+1} of an Approximate Match



The additional prefix/suffix property of a partition (see Lemma 3.9.13) suggests that an approximate match v is a subword of $y_h c_h w_{h+1} \ldots w_{h+k} c_{h+k} z_{h+k+1}$, as illustrated in Figure 6.9, 1. However, careful analysis shows that v can begin to the left of y_h. Fortunately, in that case v is a subword of $w_h c_h w_{h+1} \ldots w_{h+k-1} c_{h+k-1} z_{h+k}$, as illustrated in Figure 6.9, 2. These considerations lead to the following improved linear expected time algorithm (ILET for short):

Algorithm ILET Compute $\Psi_{lr}(t,p) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$. If $r \leq k$, then t is an interesting subword. Otherwise, for each $h, 1 \leq h \leq r-k$, let $s_h = y_h c_h \ldots w_{h+k} c_{h+k} z_{h+k+1}$. If $|s_h| \geq m-k$, then s_h is an interesting subword. Merge all overlapping interesting subwords. For each interesting subword obtained in such a way solve the k-differences problem. □

Theorem 6.8.5 Algorithm ILET correctly solves the k-differences problem.

Proof Let $edist_\delta(p,v) \leq k$ for some subword v of t which begins at position j. According to Lemma 3.9.13, there is an $l, 0 \leq l \leq k$, and a partition $\Psi = (x_1, a_1, \ldots, x_l, a_l, x_{l+1})$ of v w.r.t. p such that x_1 is a prefix and x_{l+1} is a suffix of p. Moreover, $mp_{h-1} < j \leq mp_h$ for some $h, 1 \leq h \leq r+1$. It suffices to show that v is a subword of some interesting subword. Consider the following cases:

- If $r \leq k$, then t is an interesting subword and v is a subword of t.
- If $r > k$ and $h > r-k$, then v is a subword of $w_h c_h \ldots w_r c_r w_{r+1}$ which is a subword of $s_{r-k} = y_{r-k} c_{r-k} \ldots w_r c_r z_{r+1} = y_{r-k} c_{r-k} \ldots w_r c_r w_{r+1}$. Hence, v is a subword of s_{r-k}. Since $|v| \geq m-k$, we conclude $|s_{r-k}| \geq m-k$. Thus, s_{r-k} is an interesting subword.
- If $r > k$ and $h \leq r-k$, then mp_{h+i} is defined for all $i, 0 \leq i \leq k+1$. Let $q = mp_h - |y_h|$, that is, q points to the first character of the subword $y_h c_h$. The following subcases can occur:

  1. Suppose $q \leq j$. This case is illustrated in Figure 6.9, 1. One shows by induction on i that $mp_{h+i} \geq j + |x_1 a_1 \ldots x_i|$ holds for all $i, 0 \leq i \leq l+1$. Next we show that

  $$mp_{h+l} + |z_{h+l+1}| + 1 \geq j + |v| \tag{6.11}$$

  holds. Note that $h+l+1 \leq h+k+1 \leq r+1$, that is, z_{h+l+1} is defined. If $mp_{h+l} + 1 \geq j + |v|$, then (6.11) immediately follows. Suppose $mp_{h+l} + 1 < j + |v|$.

Since $mp_{h+l}+1 \geq j+|x_1a_1 \ldots x_la_l|$, there is a prefix u of x_{l+1} such that $mp_{h+l}+1 = j + |x_1a_1 \ldots x_la_lu|$. Let u' be such that $x_{l+1} = uu'$. Obviously, u' is a suffix of p since x_{l+1} is. Furthermore, u' is a prefix of z_{h+l+1}. Hence, we can conclude $mp_{h+l}+|z_{h+l+1}|+1 = j+|x_1a_1 \ldots x_la_lu|+|z_{h+l+1}|+1 \geq j+|a_1a_1 \ldots x_la_lu|+|u'| = j+|v|$. Thus, (6.11) holds and we derive $mp_{h+k} + |z_{h+k+1}| + 1 \geq mp_{h+l} + |z_{h+l+1}| + 1 \geq j + |v|$. Therefore, v is a subword of $s_h = y_hc_hw_{h+1} \ldots w_{h+k}c_{h+k}z_{h+k+1}$. Since $|v| \geq m-k$, we have $|s_h| \geq m-k$. Hence, s_h is an interesting subword.

2. Suppose $j < q$. This case is illustrated in Figure 6.9, 2. Assume $h = 1$. Then $q = mp_h - |y_h| = mp_h - |w_h| = 1$ which is a contradiction since $mp_0 = 0 < j < q$. Therefore, $h > 1$. Next we show by induction on i that

$$mp_{h+i-1} \geq j + |x_1a_1 \ldots x_i| \tag{6.12}$$

holds for all $i, 1 \leq i \leq l$. To establish the induction start for (6.12), suppose $mp_h < j+|x_1|$. Then $u = t_j \ldots t_{mp_h-1}$ is a prefix of x_1. Hence, u is a prefix of p since x_1 is. Moreover, u is a suffix of w_h. Since $j < q$, we have $|u| = mp_h - j > mp_h - q = |y_h|$. This is a contradiction since y_h is the longest suffix of w_h that is a prefix of p. Thus, $mp_h \geq j + |x_1|$. The induction step for proving (6.12) is straightforward and we omit it here. In analogy to case 1, one shows $mp_{h+k-1} + |z_{h+k}| + 1 \geq mp_{h+l-1} + |z_{h+l}| + 1 \geq j + |v|$. Thus, v is a subword of $w_hc_h \ldots w_{h+k-1}c_{h+k-1}z_{h+k}$ which is a subword of $s_{h-1} = y_{h-1}c_{h-1}w_hc_h \ldots w_{h+k-1}c_{h+k-1}z_{h+k}$. Therefore, v is a subword of s_{h-1}. Since $|s_{h-1}| \geq |v| \geq m-k$, we conclude that s_{h-1} is an interesting subword. □

Algorithm ILET additionally has to compute the lengths of y_h and z_h for $h, 1 \leq h \leq r+1$. Let T be the compact suffix tree for $p\$$ with the annotations *isprefix* and *issuffix* (see Definitions 4.4.5 and 4.5.2). The length of the submatch w_h is determined by scanning T from the *root*. For each location $loc_T(u)$ visited during such a scan one checks in constant time if the prefix u of w_h is a suffix of p. The longest such u is z_h. Thus, $|z_h|$ can be computed at virtually no cost. The length of y_h can be obtained from $loc = loc_T(w_h)$. This is accomplished by computing the locations of the suffixes of w_h using the function *linkloc*. For each visited location $loc_T(u)$, one checks in constant time for the suffix u of w_h if u is a prefix of p. The longest such u is y_h. Thus, $|y_h|$ can be computed in $\mathcal{O}(|w_h| - |y_h|)$ steps. Altogether, the preprocessing for ILET takes $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ time. Recall that this result was also obtained Algorithm LET. Since Algorithm ILET applies a stronger filter, its search phase is always faster than the search phase of Algorithm LET. Hence, the efficiency results of Theorem 6.8.3 hold for Algorithm ILET, too. A more precise analysis of the expected case behavior is subject to future work.

6.8.3 Sublinear Expected Time Algorithm

The sublinear expected time algorithm (SET for short) is a variation of Algorithm LET. It divides the input string into non-overlapping subwords of length $len = \lfloor (m-k)/2 \rfloor$. Since an approximate match v is of length $\geq m-k$, it contains the whole of at least one subword $t_{i \cdot len+1} \ldots t_{i \cdot len+len}$ for some $i \geq 1$. Moreover, v contains at most the first k marked characters of the partition $\Psi_{lr}(t_{i \cdot len+1} \ldots t_n, p)$. Thus, the subword $t_{i \cdot len+1} \ldots t_{i \cdot len+len}$ can be discarded if the first $k+1$ marked characters in $\Psi_{lr}(t_{i \cdot len+1} \ldots t_n, p)$ occur in $t_{i \cdot len+1} \ldots t_{i \cdot len+len}$.

Algorithm SET [CL94] Let $len = \lfloor (m-k)/2 \rfloor$ and for each $i \geq 1$ define $q_i = i \cdot len + 1$. For each $q_i \leq n$, compute $r_i = |w_1 c_1 \dots w_k c_k w_{k+1}|$ where $w_1, c_1, \dots, w_k, c_k, w_{k+1}$ are the first $2 \cdot k + 1$ items of the partition $\Psi_{lr}(t_{q_i} \dots t_n, p)$. If $r_i \geq len$, then $s_i = t_{\max\{1, q_i - \lceil (m+3 \cdot k)/2 \rceil\}} \dots t_{q_i + r_i - 1}$ is an interesting subword. Merge all overlapping interesting subwords. For each interesting subword obtained in such a way solve the k-differences problem. □

Theorem 6.8.6 Algorithm SET correctly solves the k-differences problem.

Proof Let $edist_\delta(p, v) \leq k$ for some subword v of t which begins at position j. It suffices to show that v is a subword of an interesting subword. Since $|v| \geq m - k$, there is a $q_i, 1 + len \leq q_i \leq n$ such that $t_{q_i} \dots t_{q_i + len - 1}$ is a subword of v. Let $w_1, c_1, \dots, w_k, c_k, w_{k+1}$ be the first $2 \cdot k + 1$ items of the partition $\Psi_{lr}(t_{q_i} \dots t_n, p)$ and suppose $r_i = |w_1 c_1 \dots w_k c_k w_{k+1}|$. In analogy to Theorem 6.8.2, one shows $r_i \geq |v|$. Since $|v| \geq len$, we obtain $r_i \geq len$. Hence, s_i is an interesting subword. Moreover, $m + k \geq |v|$ implies $j \geq q_i + len - |v| \geq \max\{1, q_i + len - (m + k)\} \geq \max\{1, q_i - \lceil (m + 3 \cdot k)/2 \rceil\}$. Hence, v is a subword of s_i. □

Let k^* be as in Theorem 6.8.3. In [CL94], it is shown that on the average only the first $q = (k + 1) \cdot (\log_{|\mathcal{A}|} m + \mathcal{O}(1))$ characters of each subword $t_{q_i} \dots t_{q_i + len - 1}$ are examined by Algorithm SET, provided $k < k^*/2 - 3$. Since there are $2 \cdot n/(m - k)$ of these subwords, the total number of examined characters is $2 \cdot n \cdot q/(m-k)$. For this reason, Chang and Lawler call their algorithm sublinear. However, for achieving sublinearity, the following requirements must be fulfilled:

1. The length of the submatches $w_1, \dots, w_{k+1}$ must directly be determined by scanning $cst(p\$)$ while reading $t_{q_i} \dots t_n$ character by character (see section 3.9.2). This takes $\mathcal{O}(\sum_{i=1}^{k+1} |w_i|)$ time. If the matching statistics of t w.r.t. p would be used for this, then only $\mathcal{O}(k)$ steps are necessary. However, then the algorithm would not be sublinear, because $\mathcal{O}(|\mathcal{A}_p| \cdot (m + n))$ preprocessing time would be required to determine the matching statistics.

2. The suffix of $t_{q_i} \dots t_{q_i + len - 1}$ which is not examined for computing the length of the submatches $w_1, \dots, w_{k+1}$ must be skipped in constant time. This requires the input string to be given as an array of size n. Note that Chang and Lawler assume that the input string is given online [CL94, page 328].

From the first point and a previous remark (see page 187), we conclude that the matching statistics in Algorithm SET and Algorithm LET is necessary only when the checking of the interesting subwords for approximate matches is done by Chang and Lawler's version of the Diagonal Transition Algorithm. From the second point we conclude that Algorithm SET is not truly sublinear. This fact was already remarked but not explained by Myers [Mye94a].

Recently, Chang and Marr [CM94], have described a variation of Algorithm SET, called optimal sublinear expected time algorithm (OPT for short). This variation also divides the input string into non-overlapping subwords of length $\lfloor (m-k)/2 \rfloor$. But instead of computing maximal matches, Algorithm OPT looks up precomputed information about approximate matches between all strings $w \in \mathcal{A}^q$ and all subwords of p. ($q = r \cdot \log_{|\mathcal{A}|} m$ where r is a constant which is independent of m and n, but dependent on $|\mathcal{A}|$.) This means that the preprocessing for Algorithm OPT is independent of the input string. For complexity results we refer the reader to [CM94].

6.8.4 Improved Sublinear Expected Time Algorithm

There are two sources for improving Algorithm SET. First, it is not necessary to inspect the entire stretch of $\lceil (m + 3 \cdot k)/2 \rceil$ characters to the left of position q_i. It suffices to include the subword $t_{q_i - len} \ldots t_{q_i - 1}$ immediately to the left of q_i. Second, one can exploit the fact that there is always a partition of an approximate match ending with a suffix of p. Hence, it suffices to consider the subword $|w_1 c_1 \ldots w_k c_k z_{k+1}|$ where z_{k+1} is the longest prefix of w_{k+1} that is a suffix of p. These considerations lead to the following improved sublinear expected time algorithm (ISET for short):

Algorithm ISET Let len and q_i be defined as in Algorithm LET. For each $q_i \leq n$ compute $r_i = |w_1 c_1 \ldots w_k c_k z_{k+1}|$ where $w_1, c_1, \ldots, w_k, c_k, w_{k+1}$ are the first $2 \cdot k + 1$ items of the partition $\Psi_{lr}(t_{q_i} \ldots t_n, p)$ and z_{k+1} is the longest prefix of w_{k+1} that is a suffix of p. If $r_i \geq len$, then $s_i = t_{q_i - len} \ldots t_{q_i + r_i - 1}$ is an interesting subword of t. Merge all overlapping interesting subwords. For each interesting subword obtained in such a way solve the k-differences problem. □

Theorem 6.8.7 Algorithm ISET correctly solves the k-differences problem.

Proof Let $edist_\delta(p, v) \leq k$ for some subword v of t which begins at position j. It suffices to show that v is a subword of an interesting subword. There is an $i \geq 1$ such that $q_i - len \leq j \leq q_i - 1$ and $t_{q_i} \ldots t_{q_i + len - 1}$ is a subword of v. Let $w_1, c_1, \ldots, w_k, c_k, w_{k+1}$ be the first $2 \cdot k + 1$ items of the partition $\Psi_{lr}(t_{q_i} \ldots t_n, p)$ and suppose $r_i = |w_1 c_1 \ldots w_k c_k z_{k+1}|$ where z_{k+1} is the longest prefix of w_{k+1} that is a suffix of p. In analogy to Theorem 6.8.5, one shows $r_i \geq |v|$. Since $|v| \geq len$, we conclude $r_i \geq len$. Hence, $s_i = t_{q_i - len} \ldots t_{q_i + r_i - 1}$ is an interesting subword. Obviously, v is a subword of s_i. □

The computation of the lengths of the submatches can be done as in Algorithm ILET. Hence, z_{k+1} is computed at virtually no cost, and ISET determines the interesting subwords of t as efficient as Algorithm SET does. Since Algorithm ISET applies a stronger filter, its search phase is always faster than the search phase of Algorithm SET. A more precise analysis of the expected case behavior is subject to future work.

6.8.5 Implementation

Let $\Psi_{lr}(t, p) = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$. Suppose T is the compact suffix tree of $p\$$ with the annotations, *depth*, *isprefix*, and *issuffix*, as used in section 4.5. Recall that the latter two annotations enable us to decide in constant time for each location $loc_T(u)$ in T if u is a prefix of p or if u is a suffix of p, respectively. Assume $loc = loc_T(w_h)$ for some $h, 1 \leq h \leq r + 1$ and let *vnodes* be the list of nodes visited while traversing T from the root to *loc*. *loc*2*len* computes the triple $(wlen, ylen, zlen)$ where $wlen = |w_h|$, $ylen$ is the length of the longest suffix of w_h that is a prefix of p, and $zlen$ is the length of the longest prefix of w_h that is a suffix of p.

```
loc2len::([ctree α (num,bool,bool)],location α (num,bool,bool))→(num,num,num)
loc2len (vnodes,loc)
  = (wlen,ylen,zlen)
    where wlen = locdepth loc
          slocs = suffixlocs (:[]) loc
          ylen = hd [d | (d,loc')←zip2 [wlen,wlen-1..] slocs; locisprefix loc']
          zlen = wlen,                                          if locissuffix loc
               = last [d | N es link (d,isp,True)←vnodes], otherwise
```

Since $wlen = |loc|$, we obtain $wlen$ directly from loc using the function $locdepth$. To calculate $ylen$, the list $slocs$ of the locations of the suffixes of w_h is computed. $ylen$ is the depth of the first location loc' in $slocs$ which corresponds to a prefix of p. The latter condition is checked by the function $locisprefix$. The laziness guarantees that only $wlen - ylen + 1$ elements of $slocs$ are evaluated. Hence, $ylen$ is computed in $\mathcal{O}(wlen - ylen)$ steps. Note that $slocs$ contains the location of ε. Since ε is a prefix of p, $ylen$ is welldefined. If $(locissuffix\ loc)$ is true, then w_h is a suffix of p and therefore $zlen = wlen$. Otherwise, $zlen$ is the depth of the last node in $vnodes$ which is of the form $(N\ es\ link\ (d, isp, True))$, where the value $True$ in the tag means that the node location corresponds to a suffix of p. Since ε is a prefix of w_h, $zlen$ is welldefined. It is easily verified that $loc2len$ needs constant space and $\mathcal{O}(|w_h|)$ time to compute the triple $(wlen, ylen, zlen)$.

The function *preprocess* computes the triple

$$(mps, ylens, zlens) = ([mp_0, mp_1, \ldots, mp_r, mp_{r+1}], [|y_1|, \ldots, |y_{r+1}|], [|z_1|, \ldots, |z_{r+1}|])$$

where the mp_h, y_h, and z_h are as in Definitions 6.8.1 and 6.8.4.

```
preprocess::α→(string α)→num→(string α)→([num],[num],[num])
preprocess sentinel p m t
  = (mps,ylens,zlens)
    where loceps = LocN (cst (p++[sentinel],m+1) (depthisprefixissuffix m))
          wyzlens = map loc2len (lrpartition loceps t)
          wlens = map first wyzlens
          ylens = hd wlens:map second (drop 1 wyzlens)
          zlens = map third (init wyzlens) ++ [last wlens]
          mps = 0:[1 + mp + wlen | (mp,wlen)←zip2 mps wlens]
```

First *preprocess* computes T and the location $loceps = loc_T(\varepsilon)$. This takes $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time and $\mathcal{O}(m)$ space. The three annotations of T are calculated in $\mathcal{O}(m)$ by the annotation function *depthisprefixissuffix* as defined in section 4.5.1. *preprocess* calls the function *lrpartition* to compute a list of pairs $(vnodes_h, loc_h)$ where $loc_h = loc_T(w_h)$ and $vnodes_h$ is the list of nodes visited while traversing T from the *root* to loc_h. From each pair of this list the triple $(wlen, ylen, zlen)$ is obtained using the function $loc2len$. This yields a list $wyzlens$ in $\mathcal{O}(|\mathcal{A}_p| \cdot n)$ steps. $wlens$, $ylens$, and $zlens$ are computed by selecting the *first*, the *second*, and the *third* element of the triples in $wyzlens$, respectively. Note that $y_1 = w_1$ and $z_{r+1} = w_{r+1}$ according to Definition 6.8.4. Therefore, the first element in $ylens$ is set to the first element in $wlens$ and the last element in $zlens$ is set to the last element in $wlens$. mps is the list of running sums obtained by adding up the values in $wlens$. We use the technique of programming with unknowns for computing mps. That is, mps is defined

and used simultaneously. This mandates a lazy evaluation strategy. Altogether, *preprocess* achieves a running time of $\mathcal{O}(|\mathcal{A}_p| \cdot (m+n))$ using $\mathcal{O}(m)$ space.

An interesting subword $t_i \dots t_j$ of t is represented by the pair (i,j). To implement Algorithm LET, we use the function *letisubwords* which computes the corresponding list of interesting subwords.

```
letisubwords::num→num→[num]→[(num,num)]
letisubwords k m mps
  = [(1,last mps-1)],                                          if mps' = []
  = [(mp+1,mp'-1) | (mp,mp')←zip2 mps mps'; m-k < mp'-mp], otherwise
    where mps' = drop (k+2) mps
```

For each $h, 1 \le h \le r-k$, the pair $(mp, mp') = (mp_{h-1}, mp_{h+k+1})$ is computed by zipping mps and $mps' = [mp_{k+2}, mp_{k+3}, \dots, mp_{r+1}]$. Obviously, $t_{mp+1} \dots t_{mp'-1}$ is the subword $s_h = w_h c_h \dots w_{h+k} c_{h+k} w_{h+k+1}$. If $m-k < mp'-mp$, then $|s_h| \ge m-k$. Hence, s_h is an interesting subword. Therefore, the pair $(mp+1, mp'-1)$ belongs to the list returned by *letisubwords*. Suppose $|mps| \le k+2$. Then $mps' = [\,]$ and there are less than k marked characters in t. This means that t is the only interesting subword. Therefore, *letisubwords* returns the list $[(1,n)]$ where $n+1$ is the last element of the list mps. The running time of *letisubwords* is proportional to the length of mps. Note that due to the laziness only $\mathcal{O}(k)$ elements of mps are stored at any time.

To implement Algorithm ILET, we use the function *iletisubwords* which computes the corresponding list of interesting subwords.

```
iletisubwords::num→num→([num],[num],[num])→[(num,num)]
iletisubwords k m (mps,ylens,zlens)
  = [(1,last mps-1)],                                              if zlens' = []
  = [(left,right) | (left,right)←lrpos; m-k <= right-left+1], otherwise
    where mps' = drop (k+1) mps
          zlens' = drop (k+1) zlens
          lrpos = [(mp-ylen,mp'+zlen) |
                    (ylen,mp,mp',zlen)←zip4 ylens (drop 1 mps) mps' zlens']
```

For each $h, 1 \le h \le r-k$ the 4-tuple $(ylen, mp, mp', zlen) = (|y_h|, mp_h, mp_{h+k}, |z_{h+k+1}|)$ is computed by zipping the following four lists:

$$ylens \quad [mp_1, mp_2, \dots, mp_{r+1}] \quad [mp_{k+1}, mp_{k+2}, \dots, mp_{r+1}] \quad [|z_{k+2}|, |z_{k+3}|, \dots, |z_{r+1}|]$$

Let $left = mp - ylen$ and $right = mp' + zlen$. Obviously, $t_{left} \dots t_{right}$ is the subword $s_h = y_h c_h \dots w_{h+k} c_{h+k} z_{h+k+1}$. If $m-k \le right - left + 1$, then s_h is an interesting subword and the pair $(left, right)$ belongs to the list returned by *iletisubwords*. Note that $|mps| \le k+2$ if and only if $zlens' = [\,]$. Hence, if $zlens'$ is empty, then t is the only interesting subword and *iletisubwords* returns a list with the only element $(1,n) = (1, last\ mps - 1)$.

The function *setisubwords* computes the list of interesting subwords according to Algorithm SET. At first, $len = \lfloor (m-k)/2 \rfloor$ and $toleft = \lceil (m + 3 \cdot k)/2 \rceil$ are determined. Then the function *divide* is called to construct the list

$$qslist = [(q_i, t_{q_i} \dots t_n) \mid q_i = 1 + i \cdot len, 1 \le i, q_i \le n]$$

Since $t_{q_i} \ldots t_n$ is not evaluated, this takes $\mathcal{O}(n)$ time. For each pair (q, s) occurring in *qslist*, a function *howfar* is called which calculates how far the first $k+1$ maximal matches in s reach. More precisely, *howfar* determines the first $2 \cdot k+1$ items $w_1, c_1, \ldots, w_k, c_k, w_{k+1}$ of the partition $\Psi_{lr}(s, p)$ and returns the length r of $w_1 c_1 \ldots w_k c_k w_{k+1}$. Note that *lrpartition* only evaluates $k+1$ locations and determines the length of the submatches from these. Hence, *howfar* takes $\mathcal{O}(|\mathcal{A}_p| \cdot r)$ time. If $r \geq len$, then $t_{\max\{1,q-toleft\}} \ldots t_{q+r-1}$ is an interesting subword.

```
setisubwords::α→num→(string α)→num→(string α)→[(num,num)]
setisubwords sentinel k p m t
  = [(max2 1 (q-toleft),q+r-1) | (q,s)←qslist; r←[howfar s]; len <= r]
    where len = (m-k) div 2
          toleft = (1 + m + 3 * k) div 2
          qslist = zip2 [1+len,1+len+len..] (divide len (drop len t))
          loceps = LocN (cst (p++[sentinel],m+1) (depthisprefixissuffix m))
          howfar s = k + sum (take (k+1) [wlen | loc←lrpartition loceps s;
                                               (wlen,ylen,zlen)←[loc2len loc]])

divide::num→(string α)→[string α]
divide q [] = []
divide q s = s:divide q (drop q s)
```

The function *isetisubwords* computes the list of interesting subwords according to Algorithm ISET. It is very similar to *setisubwords*. The main difference is that the local function *howfar* sums up the length of the items $w_1, c_1, \ldots, w_k, c_k, z_{k+1}$ where z_{k+1} is the longest prefix of w_{k+1} that is a suffix of p.

```
isetisubwords::α→num→(string α)→num→(string α)→[(num,num)]
isetisubwords sentinel k p m t
  = [(q-len,q+r-1) | (q,s)←qslist; r←[howfar s]; len <= r]
    where len = (m-k) div 2
          loceps = LocN (cst (p++[sentinel],m+1) (depthisprefixissuffix m))
          qslist = zip2 [1+len,1+len+len..] (divide len (drop len t))
          howfar s = sumup k (map loc2len (lrpartition loceps s))

sumup::num→[(num,num,num)]→num
sumup (l+1) ((wlen,ylen,zlen):rest) = 1 + wlen + sumup l rest
sumup 0 ((wlen,ylen,zlen):rest) = zlen
sumup l rest = 0
```

mergeisubwords merges consecutive interesting subwords $(begin, end)$ and $(begin', end')$ to one subword $(begin, end')$ whenever they overlap or follow each other without a gap. This takes time proportional to the number of interesting subwords.

```
mergeisubwords::[(num,num)]→[(num,num)]
mergeisubwords ((begin,end):(begin',end'):isubwords)
  = mergeisubwords ((begin,end'):isubwords),              if end+1 >= begin'
  = (begin,end):mergeisubwords ((begin',end'):isubwords), otherwise
mergeisubwords isubwords = isubwords
```

($pairs2strings\ 1\ t$) is applied to the list of merged interesting subwords. It returns for every interesting subword $(begin, end)$ the triple $(begin, end, s)$ where s is a lazy list representing the suffix $t_{begin} \ldots t_n$ of t. This takes $\mathcal{O}(n)$ steps.

```
pairs2strings::num→(string α)→[(num,num)]→[(num,num,string α)]
pairs2strings j s ((begin,end):isubwords)
  = (begin,end,s'):pairs2strings begin s' isubwords
    where s' = drop (begin-j) s
pairs2strings j s [] = []
```

The four filter algorithms differ only in the way interesting subwords are computed. We therefore introduce a higher order function *appfilter* which abstracts from this difference. *appfilter* takes two functions *filterkdiff* and *solvekdiff* as arguments. *filterkdiff* determines the interesting subwords in the input string t. These are merged. Then for each merged subword $(begin, end)$, the corresponding suffix s of t which begins at position $begin$ is computed. This gives a list $misubwords$ of triples. For each element $(begin, end, s)$ in $misubwords$, the function *solvekdiff* is called to solve the k-differences problem for the subword $t_{begin} \ldots t_{end}$.

```
appfilter::(string α→[(num,num)])→((num,num,string α)→[num])→(string α)→[num]
appfilter filterkdiff solvekdiff t
  = [j | (begin,end,s)←misubwords; j←solvekdiff (begin,end,s)]
    where misubwords = pairs2strings 1 t (mergeisubwords (filterkdiff t))
```

Implementations of the four filter Algorithms LET, ILET, SET, and ISET are obtained in a straightforward way by supplying *appfilter* with the specific function for computing the interesting subwords. This resembles the close relationship of the four algorithms. For convenience we introduce the following type synonym:

```
filteralgorithm α == ((num,num,string α)→[num])→
                     α→num→(string α)→(string α)→[num]

appLET::filteralgorithm α
appLET solvekdiff sentinel k p
  = appfilter (letisubwords k m.first.preprocess sentinel p m) solvekdiff
    where m = #p

appILET::filteralgorithm α
appILET solvekdiff sentinel k p
  = appfilter (iletisubwords k m.preprocess sentinel p m) solvekdiff
    where m = #p

appSET::filteralgorithm α
appSET solvekdiff sentinel k p = appfilter (setisubwords sentinel k p m) solvekdiff
                                 where m = #p

appISET::filteralgorithm α
appISET solvekdiff sentinel k p = appfilter (isetisubwords sentinel k p m) solvekdiff
                                  where m = #p
```

Every of the four implementations above does not use a specific function to solve the k-differences problems for the interesting subwords. We leave it to the application to derive instances with a fixed "k-differences subproblem solver". From the remarks above, it is clear that the implementations are optimal. *appSET* and *appISET* are not sublinear since they call the function *divide* to compute the list *qslist*. This takes $\mathcal{O}(n)$ steps. However, an imperative implementation which reads t online cannot not achieve a better than linear asymptotic runtime. Note that in the worst case, the submatches are of length $\mathcal{O}(m)$. Hence, *appSET* and *appISET* perform $\mathcal{O}(|\mathcal{A}_p| \cdot k \cdot m)$ steps for each subword $t_{q_i} \dots t_n$. Thus, their worst case running time is $\mathcal{O}(|\mathcal{A}_p| \cdot k \cdot m \cdot n/(m-k))$.

6.8.6 Other Approaches

By generalizing the Boyer-Moore-Horspool Algorithm, Tarhio and Ukkonen [TU93] obtained another filter algorithm for solving the k-differences problem. To explain this algorithm, we redefine the notion of minimizing edges (see page 75). Consider the edit graph $G(p,t)$. An edge $(i',j') \xrightarrow{a \to b} (i,j)$ in $G(p,t)$ is *minimizing* if $D(i,j)$ equals $D(i',j') + \delta(a \to b)$ where δ is the unit cost function. A *successful minimizing path* is any path from some node $(0,j')$ to some node (m,j) such that $D(m,j) \leq k$. The algorithm of Tarhio and Ukkonen is based on the following observation. If a successful minimizing path goes through the node $(i,h+i)$ for some $i, 0 \leq i \leq m$, then for at most k indices $l, 1 \leq l \leq m$ the character t_{h+l} is bad, that is, $t_{h+l} \neq p_r$ for all $r, \max\{1, l-k\} \leq r \leq \min\{m, l+k\}$ (see [TU93, Lemma 3]). This leads to the following method for determining interesting subwords of t. Let $h = 0$ and align p with $t_1 \dots t_m$. Perform the following step until $h > n$: For $l = m, m-1, \dots, k+1$ check if t_{h+l} is bad. If $\leq k$ bad characters are found in this way, then $t_{\max\{1,h-k\}} \dots t_{\min\{n,h+k\}}$ is an interesting subword of t. h is incremented by r and p is shifted r positions to the right where r is minimal such that at least one of the characters $t_{h+m}, t_{h+m-1}, \dots, t_{h+m-k}$ matches the corresponding pattern character. To determine r and to decide if t_{h+l} is bad, $\mathcal{A}$, p, and k are preprocessed in $\mathcal{O}((k+|\mathcal{A}|) \cdot m)$ time and $\mathcal{O}(|\mathcal{A}| \cdot m)$ space. The worst case running time of the algorithm is $\mathcal{O}(m \cdot n/k + m \cdot n)$. It occurs if t is an interesting subword. If $2 \cdot k + 1 < |\mathcal{A}|$, then the algorithm takes

$$\mathcal{O}\left(\frac{|\mathcal{A}|}{|\mathcal{A}| - 2 \cdot k} \cdot k \cdot n \cdot \left(\frac{k}{|\mathcal{A}| + 2 \cdot k^2} + \frac{1}{m}\right)\right)$$

expected time to determine the interesting subwords. If the pattern and the input alphabet is large and the threshold value is small, then the dynamic programming steps for checking if an interesting subword contains an approximate match, are executed very seldom. Hence, in this case the above bound also holds for the entire algorithm. Experimental results in [TU93] show that Tarhio and Ukkonen's algorithm runs considerably faster than Algorithm UKKco, provided the input alphabet is large.

Myers [Mye94a] has developed an algorithm for solving the k-differences problem, whose search phase is sublinear in n. To explain Myers' algorithm, we introduce the following notion: The *condensed k-neighborhood* of $s \in \mathcal{A}^*$, denoted by $neighborhood_k(s)$, is the set of strings w such that $edist_\delta(w,s) \leq k$, and for all prefixes v of w we have $v \notin neighborhood_k(s)$. A simple method to solve the k-differences problem is to construct $neighborhood_k(p)$ and to look up each $w \in neighborhood_k(p)$ in an inverted index of the input string. Unfortunately, this simple method is impractical in general since the size of the neighborhood rapidly grows

with the pattern length and the threshold value. Myers has found a way to avoid this problem by dividing p into non-overlapping subwords of length $q = \log_{|\mathcal{A}|} n$. For each such subword s the algorithm computes $neighborhood_{k'}(s)$ where $k' < k$ is a reduced threshold value. The elements in $neighborhood_{k'}(s)$ are looked up in an inverted index for the input string. Using a "doubling trick", the resulting partial approximate matches are then successively combined to obtain approximate matches with p if they exist. This is technically complicated, and we do not give the details here. Myers has shown that, if the input alphabet is fixed and finite, the preprocessing of t into a suitable inverted index can be accomplished in $\mathcal{O}(n)$ space and time. The inverted index only depends on $\mathcal{A}$ and t, but not on p. Hence, it can be used for varying patterns. After precomputing the inverted index, the approximate matches are found in $\mathcal{O}(k \cdot n^{f(k/m)} \cdot \log n)$ time where f is a monotone increasing function such that $f(0) = 0$. The search phase of the algorithm is sublinear if k/m is small enough to guarantee that $f(k/m) < 1$. As shown in [Mye94a], this is true for a wide range of values. For instance, if $|\mathcal{A}| = 4$, then $f(k/m) < 1$ for all $k/m < 0.33$. If $|\mathcal{A}| = 20$, then $f(k/m) < 1$ for all $k/m < 0.56$. Practical experiments show that Myers' algorithm is sometimes much faster than Algorithm UKKco and the algorithm of Wu, Manber, and Myers [WMM92] which we described in section 6.5.

6.9 Overview of the Implementations

In this chapter, we presented the most important algorithms for solving the approximate string searching problem. We implemented most of the algorithms. Figure 6.10 gives an overview of the programs. Except for DTpp, all functional implementations are optimal. That is, they achieve the same asymptotic efficiency as their imperative counterparts. Input strings are scanned online from left to right. At each time the access to t is either restricted to one input character or to a small sliding window of size $\mathcal{O}(m)$ or $\mathcal{O}(k \cdot m)$. A remarkable virtue of our functional implementations is that an explicit buffering mechanism is not required to achieve space efficiency. Lazy evaluation and the memory management of the Miranda system guarantee that only the needed portion of the input string is stored at any time. Note that the programs reuse several parts of our string processing machinery. Let us only enumerate the most important parts:

- For *appSEL*, *appSELco*, *appSESA*, and *appUKKco* we have developed table specifications, reusing the function *absnextcol* to realize the main recurrences of the Wagner-Fischer Algorithm.
- For *appSEL*, *appSELco*, and *appUKKco* we have reused the function *scanl* to implement the search phase.
- For the implementation of Algorithm MS (see function *mstats*) we have reused the functions *scanprefix* and *linkloc* to realize the traversing of the suffix tree.
- For *appLET*, *appILET*, *appSET*, *appISET*, and *appDTpp* we have reused the function *cst* to compute compact suffix trees.
- For *appLET*, *appILET*, *appSET*, and *appISET* we have reused the function *lrpartition* to compute left-to-right partitions in different ways.

Figure 6.10: Overview of the app-Programs

function	running time worst	expected	comment
appSEL	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(m \cdot n)$	simplest variation of the Wagner-Fischer Algorithm; do not charge insertions into empty string; set first row to 0
appSELco	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(k \cdot n)$	cutoff variation of *appSEL*; compute distance column up to last essential index
appSESA	$\mathcal{O}(q \cdot \lvert\mathcal{A}\rvert \cdot m + n)$		incremental construction of *SES*-automaton in top down strategy; states are shortest essential suffixes; compute length and distance columns; scan t in constant time per character; extension 1 required
	$q = 2 \cdot \lvert\mathcal{A}\rvert^{m+k} - 1$	$q =$?	q is number of constructed states
appUKKco	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(k \cdot n)$	optimization of *appSELco* for unit cost function
appCdfa	$\mathcal{O}(q \cdot \lvert\mathcal{A}\rvert \cdot m + n)$		incremental construction of Column-DFA; store normalized distance columns in ternary tree; postpone processing of columns by storing them in a queue; scan t in constant time per character; extension 1 required
	$q = 3^m$	$q =$?	q is number of processed columns
appDTbf	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(k \cdot n)$	compute positions in D-diagonals where values increase; calculate jumps by pairwise character comparisons
appDTpp	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot (m+n) + m^2 + m \cdot n)$	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot (m+n) + m^2 + k \cdot n)$	variation of *appDTbf*; calculate jumps in constant time from precomputed matching statistics and table *pjump*; extension 1 required; improved $\mathcal{O}(k \cdot n)$-worst case running time of search phase not achieved
appCP	$\mathcal{O}(\lvert\mathcal{A}\rvert \cdot m + m \cdot n)$	$\mathcal{O}(\lvert\mathcal{A}\rvert \cdot m + \frac{k \cdot n}{\sqrt{\lvert\mathcal{A}\rvert}-1})$	preprocess $\mathcal{A}$ and p; compute endpoints of runs of consecutive integers in constant time; extension 1 required
appLET	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot (m+n) + m \cdot n)$	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot (m+n) + n)$ if $k < k^*$	discard subwords of t which are shorter than $m-k$ and which contain $k+1$ marked characters; compute length of submatches directly from $cst(p\$)$
appILET	see *appLET*		variation of *appLET*; stronger filter; exploits prefix/suffix property of partition
appSET	$\mathcal{O}(\lvert\mathcal{A}_p\rvert \cdot k \cdot m \cdot n/(m-k))$	$\mathcal{O}(\frac{\log_{\lvert\mathcal{A}\rvert} \cdot k \cdot n}{m-k})$ if $k < k^*/2 - 3$	divide t into non-overlapping subwords of length $(m-k)/2$; discard subword if it contains the first $k+1$ marked characters
appISET	see *appSET*		variation of *appSET*; stronger filter; exploits suffix property of partition

Note that the first three programs work for arbitrary cost functions.
All other programs are restricted to the unit cost function.

Chapter 7

Performance Results

We extensively studied the practical efficiency of the different exact and approximate string searching algorithms described in chapters 4, 5, and 6. For all measurements we used random patterns of varying lengths, and random input strings of fixed length $n = 10.000$ over alphabets of the sizes $|\mathcal{A}| = 4, 20, 50, 80$. All patterns and input strings were generated independently.

The main goal is to validate the analytical efficiency results of our implementations and to get an idea of the relative efficiency of the different programs. Rather than the runtime, we show the number of SKI-reductions (see section 2.2), as reported by the Miranda system (version 2.014). To decrease random variation, each measurement shows the average number of SKI-reductions performed by a particular program for ten different patterns of fixed length. Recall that the number of SKI-reductions refers to an abstract machine. Hence, the measurements are independent of the particular computer and the operating system. This also means that it was not necessary to perform a particular measurement more than once. We should remark that the number of SKI-reductions and the runtime correlate (see section 2.2). For example, a SPARC 10/41 performs a constant rate of about 80.000 SKI-reductions per CPU-second.

7.1 Exact String Searching

We tested the effect of the pattern length on the exact string searching algorithms BF, KMP, KR, BM, BMH, BMS, CL, and CL′. The measurements are shown in Figure 7.1. The smallest number of reductions in each row is printed in bold face. The main results are as follows:

- The performance of *exsBF* is almost independent of the pattern length and the alphabet size. This is consistent with the expected running time of $\mathcal{O}(m + n)$. *exsBF* is the fastest program for $|\mathcal{A}| = 4$ and the third fastest program for $|\mathcal{A}| \geq 20$. The good behavior may be due to the simplicity of the implementation and the fact that *exsBF* performs string comparisons with the built-in predicate "=".

- The performance of *exsKMP* is almost independent of the pattern length and improves slightly with growing alphabet size. This is consistent with the results in [BY89a]. In almost all cases, *exsKMP* is the slowest program. This may be explained by the

Figure 7.1: Number of Reductions in 1000 for the exs-Programs

$\|\mathcal{A}\|$	m	BF	KMP	KR	BM	BMH	BMS	CL	CL′
4	8	**173**	608	460	277	243	271	1299	833
4	16	**174**	635	460	311	264	266	738	581
4	32	**174**	635	459	293	283	381	470	371
4	64	**174**	634	459	319	307	345	323	304
4	128	**175**	630	458	440	314	364	234	260
20	8	170	559	460	203	171	**143**	693	457
20	16	170	561	460	167	136	**124**	473	324
20	32	170	565	459	168	121	**118**	317	244
20	64	170	560	459	208	**116**	125	228	198
20	128	170	560	458	295	**117**	123	197	205
50	8	170	542	460	193	164	**121**	595	392
50	16	170	542	460	152	125	**93**	412	276
50	32	170	544	459	153	108	**83**	300	217
50	64	170	545	459	214	102	**90**	243	201
50	128	170	543	458	383	105	**104**	244	230
80	8	170	543	460	189	160	**116**	586	389
80	16	170	543	460	150	124	**88**	423	283
80	32	170	544	459	149	107	**78**	325	230
80	64	170	543	459	215	100	**78**	268	218
80	128	170	543	458	441	103	**95**	284	280

fact that it uses the function *next* which works on arbitrary atomic $\mathcal{A}^+$-trees. We conjecture that an optimized version of *next* which is tailored for KMP-trees, leads to a considerable speed-up of *exsKMP*.

- The performance of *exsKR* is almost constant. Since m is small compared to n, this fact is consistent with the expected performance of $\mathcal{O}(m+n)$. In most cases, *exsKR* is the second slowest program. The bad performance may be due to the overhead of Miranda's unbound precision arithmetic.

- As expected, *exsBM* performs better with growing alphabet size. There is an interesting effect of the pattern length. The step from $m = 8$ to $m = 16$ leads to a speed-up (except for $|\mathcal{A}| = 4$). The step from $m = 16$ to $m = 32$ does not have an influence on the speed. The step from $m = 32$ over $m = 64$ to $m = 128$ slows down *exsBM* considerably. This may be explained by the increasing preprocessing effort for larger patterns which is not amortized by the larger expected shifts.

- The performance of *exsBMH* improves with growing alphabet size and pattern length. This is consistent with the analytical results. *exsBMH* is always faster than *exsBM*. This may be explained by the smaller preprocessing effort of *exsBMH* and the fact that *exsBMH* performs string comparisons with the built-in predicate "=". *exsBMH* beats *exsBF* for $|\mathcal{A}| \geq 20$. In most cases, *exsBMH* is the second fastest program.

- *exsBMS* shows a similar behavior as *exsBMH* w.r.t. alphabet size and pattern length. If $|\mathcal{A}| \geq 20$, then *exsBMS* is in almost all cases the fastest program. The speed advantage over the second fastest program *exsBMH* is sometimes considerable, especially for larger alphabets. This may be explained as follows. If the alphabet size is large, then

Figure 7.2: Number of Reductions in 1000 for the exms-Programs

$\|\mathcal{A}\|$	r	STS	AC	$\|\mathcal{A}\|$	r	STS	AC	$\|\mathcal{A}\|$	r	STS	AC	$\|\mathcal{A}\|$	r	STS	AC
4	32	1763	**701**	20	32	1477	**672**	50	32	1606	**680**	80	32	1550	**669**
4	64	2285	**814**	20	64	2114	**859**	50	64	2036	**857**	80	64	2038	**850**
4	128	2600	**881**	20	128	2768	**1048**	50	128	2784	**1127**	80	128	2992	**1144**
4	256	3043	**892**	20	256	3482	**1285**	50	256	4306	**1577**	80	256	4747	**1713**
4	512	3363	**915**	20	512	3865	**1461**	50	512	6221	**2000**	80	512	7376	**2376**
4	1024	3782	**951**	20	1024	4392	**1543**	50	1024	7593	**2213**	80	1024	10416	**2982**

the maximal shift value of $m+1$ for *exsBMS* and of m for *exsBMH* occurs often. To perform a maximal shift, *exsBMS* evaluates the expression $(reverse\ (take\ m\ y), drop\ m\ y)$, while *exsBMH* calls the function *rightshift* which may be much slower.

- *exsCL* behaves bad if the pattern is short and the alphabet size is small. This is consistent with the analytical results. For $|\mathcal{A}| = 4$ and $m = 128$, *exsCL* is the second fastest program. However, for $|\mathcal{A}| \geq 20$ it cannot compete with *exsBMH* and *exsBMS*.

- *exsCL'* shows the same behavior as *exsCL* w.r.t. pattern length and alphabet size. In almost all cases, there is a slight speed advantage over *exsCL* which reduces with growing alphabet size and pattern length.

Figure 7.4 shows a plot of the measurements for the five fastest programs *exsBF*, *exsBM*, *exsBMH*, *exsBMS*, and *exsCL'*. Based on the empirical results, we recommend to choose *exsBF* for $|\mathcal{A}| = 4$ and *exsBMS* for $|\mathcal{A}| \geq 20$.

7.2 Multiple Exact String Searching

The Suffix Tree Search Algorithm and the Aho-Corasick Algorithm were applied to sets of random patterns. In particular, for each $r = 32, 64, 128, 256, 512, 1024$ we randomly chose ten different sets $\mathcal{P}$ of random patterns such that $\sum_{p \in \mathcal{P}} |p| = r$. The measurements are shown in Figure 7.2. The smallest number of reductions in each row is printed in bold face. A plot of the measurements can be found in Figure 7.5.

The main results of the measurements are as follows: *exmsSTS* and *exmsAC* show the expected behavior. They both slow down with a growing alphabet size and growing r. The effect on the speed when doubling r is moderate for $|\mathcal{A}| \leq 20$, and increasing for $|\mathcal{A}| \geq 50$. This may be explained by the fact that for large alphabets random patterns do not have long common subwords. For *exmsSTS* this means that almost all patterns force the lazy suffix tree algorithm to construct a new path in the suffix tree. For *exmsAC* this means that most transitions take $\mathcal{O}(|\mathcal{A}|)$ time since they mostly occur at the *root* of the KMP-tree which has $\mathcal{O}(|\mathcal{A}|)$ outgoing edges. Our measurements reveal that *exmsAC* is always two to three times faster than *exmsSTS*. The preprocessing effort for constructing the compact suffix tree of the input string does therefore not pay off for *exmsSTS*.

7.3 Approximate String Searching

We tested the approximate string searching algorithms SESA, UKKco, Cdfa, DTbf, DTpp, CP, LET, ILET, SET, ISET. For the last four algorithms we have used Algorithm *UKKco* to solve the k-differences problems for the interesting subwords. Recall that Algorithm SESA works for arbitrary cost functions. However, we have always supplied *appSESA* with the unit cost function, in order to compare it with the other programs which are restricted to the k-differences problem. Note that we have not measured the functions *appSEL* and *appSELco* since *appUKKco* is an optimization of *appSELco* which in turn is an optimization of *appSEL*. We adopted the test scheme of [JTU91] and performed three test series:

1. We measured the effect of the pattern length in a test series with varying $m = 8$, 16, 32, 64, 128, and fixed $k = 4$.

2. We measured the effect of an absolute threshold value in a test series with varying $k = 1, 2, 3, 4$ and fixed $m = 10$.

3. We measured the effect of a relative threshold value in a test series with varying $m = 8, 16, 32, 64, 128$ and $k = m/8$.

The measurements are shown in Figure 7.3. The smallest number of reductions in each row is printed in bold face. Unfortunately, *appSESA* and *appCdfa* run into heap space problems for $m \in \{64, 128\}$ and $k = m/8$. Therefore, the corresponding measurements are missing in the third test series. The main results of the measurements are as follows:

- The performance of *appSESA* is good, except if the pattern and the threshold value is large. In the first and third test series, the speed improves with larger alphabet size. If $k = 4$ is fixed, then a growing pattern length slows down *appSESA*. For $|\mathcal{A}| = 4$ the decrease in speed is considerable, for $|\mathcal{A}| \geq 20$ it is moderate. If $m = 10$ is fixed, then a growing threshold value leads to a considerable increase in the running time. However, in many cases *appSESA* beats the other programs. We found the good relative performance of *appSESA* remarkable since it does not exploit any property of the unit cost model, like all the other algorithms do.

- Our test series validate the $\mathcal{O}(k \cdot n)$ expected running time of *appUKKco*. The first series shows that for fixed $k = 4$ a varying pattern length does (almost) not influence the speed. The second and third series show that a growing threshold value slows down the program considerably. A growing alphabet size has a positive effect on the performance of *appUKKco*. This effect was also measured in [JTU91]. Note that except for the cases where *appSESA* ran into heap space problems, it is always faster than *appUKKco*. Since the latter is an optimization of *appSELco*, we can conclude that for the unit cost function *appSESA* is faster than *appSELco*. It should be evaluated whether this relation holds for other cost functions, too.

- As expected, *appCdfa* shows a good performance for small threshold values, small patterns, and small alphabets. For instance, if $m = 10$, $k \leq 3$, and $|\mathcal{A}| \leq 20$, then *appCdfa* is the second fastest program. In most cases, *appCdfa* is slower than *appSESA*. This may be explained by the fact that *appCdfa* uses list indexing, while *appSESA* does

not. Another reason may be that the automata construction performed by *appSESA* is "more lazy", which means that the construction of a new transition does not force the algorithm to construct many new states. In order to validate this, detailed information about the lazy evaluation process is required. This can be provided by modern profiling tools for lazy functional languages (see [RW93, SPJ95]).

- The three test series clearly reveal the $\mathcal{O}(k \cdot n)$ expected running time of *appDTbf*. For fixed $k = 4$ the speed is almost invariant of the pattern length and the size of the alphabet. The second and third series show that *appDTbf* slows down considerably with growing k. In comparison to the other programs, the performance of *appDTbf* is quite bad. This may be due to the overhead created by the use of access triples. *appDTbf* is only half as fast as *appUKKco* which is also $\mathcal{O}(k \cdot n)$ in the expected case. A similar relationship was observed in [JTU91].

- *appDTpp* is the slowest program in all cases. (Note that it is also the most complicated.) The bad performance may be due to the overhead created by the use of access triples. Moreover, the preprocessing effort plays a role. Consider, for instance, the first test series. For fixed $k = 4$ there is a slight dependence on m. This may result from the precomputation of table $pjump$ which requires $\mathcal{O}(m^2)$ time.

- The first and the third test series show that *appCP* slows down by a factor 2 if m is doubled. The second test series clearly shows that the speed of *appCP* is independent of the threshold value. The larger the alphabet, the better *appCP* performs. *appCP* is one of the fastest programs for large alphabets and small patterns. This behavior is consistent with the results in [CL92, WMM92, CM94]. Note that our measurements do not validate the statement of Chang and Lampe [CL92] according to which Algorithm CP is always faster than Algorithm UKKco. Especially for large m and small alphabets, *appCP* is much slower than *appUKKco*.

- *appLET* shows the expected behavior. If $k = 4$ is fixed and m is small, then a slight absolute increase of m makes the filter much more sensitive which leads to a speed-up by a factor > 2. If m is larger, then the sensitivity of the filter does not change with growing m. If $m = 10$ is fixed, then *appLET* considerably slows down with growing threshold value and slightly speeds up with growing alphabet size. *appLET* is one of the fastest program if $k = m/8$ and m varies.

- *appILET* shows a similar behavior as *appLET* w.r.t. the threshold value, the pattern length, and the alphabet size. However, in comparison to *appLET*, the running time of *appILET* is disappointing. Although it applies a stronger filter, *appILET* is always slower than *appLET*. This can be explained by the increased preprocessing effort which does not pay off. More measurements are necessary to characterize combinations of pattern lengths and threshold values where the stronger filter of *appILET* becomes important.

- *appSET* is one of the fastest programs in the first and the third test series. The larger the pattern and the alphabet, the better *appSET* performs. If the pattern is not too small, then there is a speed advantage over *appLET*. A growing threshold value considerably slows down *appSET*. This is consistent with the theoretical results.

- In most cases, *appISET* and *appSET* show nearly identical performance. However, sometimes *appISET* runs 20 percent faster than *appSET*. These may be the cases where the stronger filter of *appISET* shows its effect. More measurements are necessary to reveal those combinations of pattern lengths and threshold values where the stronger filter of *appISET* becomes important.

On the basis of our measurements we can recommend the following choices depending on $|\mathcal{A}|$, k, and m.

1. If m and k are small, then *appSESA* is a good choice.
2. If m is small and the alphabet size is large, then *appCP* is a good choice.
3. If m is much larger than k, then *appISET* is a good choice.

Figures 7.6, 7.7, and 7.8 show the plots of the three test series for the programs *appSESA*, *appUKKco*, *appCdfa*, *appCP*, *appLET*, and *appISET*. Recall that we have no measurements for *appSESA* and *appCdfa* when $m \in \{64, 128\}$ and $k = m/8$.

If one studies our performance results, one should take into account the following facts:

- The length of the input string is relatively small. Hence, for some programs the preprocessing effort is considerable compared to the search time. More measurements with longer input strings may lead to different performance results.
- The patterns and input strings we considered are random. Measurements with real data, for instance DNA sequences or english text, may lead to different performance results.
- There are sources for optimizing the programs. Although we have several ideas for optimizations, we have not applied them since they would lead to programs which are much harder to explain and to understand, and which do not reuse many other functions. The main goal concerning efficiency was to achieve the asymptotic behavior of the corresponding imperative programs.

Figure 7.3: Number of Reductions in 1000 for the app-Programs

$\|\mathcal{A}\|$	k	m	SESA	UKKco	Cdfa	DTbf	DTpp	CP	LET	ILET	SET	ISET
4	4	8	2267	5134	**1359**	9207	14285	2286	7117	8393	15797	15902
4	4	16	**3494**	5582	8262	9274	15292	4276	7558	8861	10049	10101
4	4	32	4020	5787	9222	9290	15044	8129	**1954**	3225	6907	5422
4	4	64	4768	5967	12382	9322	15348	17043	2024	3344	**1363**	1376
4	4	128	4320	5810	10067	9293	15458	34134	2103	3347	**822**	840
20	4	8	2841	4214	2780	9114	12546	**1156**	6390	7795	10727	10786
20	4	16	3275	4269	7197	9107	13795	**2186**	2626	3548	7193	7091
20	4	32	3620	4329	6799	9101	14867	4204	2250	3584	**1550**	1568
20	4	64	3361	4313	5040	9107	15638	8317	2405	3759	**891**	902
20	4	128	3220	4306	4587	9113	16250	16415	2579	4007	**558**	572
50	4	8	1453	3943	3330	9138	11926	**764**	6306	7791	9658	9705
50	4	16	1592	3964	3558	9137	12988	**1349**	2440	3937	5729	4633
50	4	32	2168	4024	4762	9126	14485	2749	2827	4273	**1525**	1539
50	4	64	2384	4048	5530	9122	16019	5480	3079	4458	**967**	975
50	4	128	2165	4019	4972	9124	17237	10868	3302	4627	**691**	700
80	4	8	1536	3931	3903	9136	11819	**705**	6370	7864	9565	9609
80	4	16	1548	3934	3073	9136	12953	**1191**	2624	4135	5129	4087
80	4	32	**1618**	3943	3470	9136	14707	2295	3353	4826	1656	1669
80	4	64	1584	3940	2734	9136	16774	4408	3982	5394	**1118**	1126
80	4	128	1637	3944	3503	9135	18738	8781	4286	5635	**845**	853
4	1	10	433	2289	**423**	3173	8109	2658	2263	3159	4392	3272
4	2	10	**668**	3235	677	5195	10364	2658	4847	4982	6809	6870
4	3	10	**1273**	4304	1490	7233	12616	2659	6316	7607	10548	10626
4	4	10	**2418**	5358	2726	9252	14852	2660	7370	8660	13080	13156
20	1	10	522	1925	**470**	3110	6458	1459	1939	3365	1600	1534
20	2	10	1061	2687	**766**	5105	8611	1459	2075	3376	4385	3444
20	3	10	2033	3493	1509	7113	10765	**1459**	4599	5016	7413	7458
20	4	10	3407	4292	3675	9101	12906	**1459**	6515	7874	9117	9159
50	1	10	**503**	1840	538	3102	5779	909	2103	3615	1269	1312
50	2	10	**774**	2552	917	5113	7902	909	2104	3615	2414	2085
50	3	10	1239	3269	1733	7122	10022	**909**	2553	3785	6652	6687
50	4	10	1936	3990	3086	9127	12138	**909**	5663	6914	8139	8171
80	1	10	**497**	1825	552	3101	5749	815	2195	3728	1269	1316
80	2	10	**712**	2526	857	5115	7869	815	2195	3728	2136	1981
80	3	10	1074	3230	1702	7126	9985	**815**	2366	3779	6627	6660
80	4	10	1595	3939	3561	9135	12100	**815**	5343	6566	8103	8133
4	1	8	455	2408	**438**	3202	7747	2270	3217	3413	5348	4836
4	2	16	**762**	3451	778	5219	10804	4276	2432	3318	5594	4022
4	4	32	4020	5787	9222	9290	15044	8129	**1954**	3225	6907	5422
4	8	64		10149		17366	24301	17043	2065	3341	12715	12671
4	16	128		18655		33503	42328	34134	2103	3341	21482	21487
20	1	8	525	1902	**474**	3105	6115	1156	1855	3289	2577	2001
20	2	16	1071	2681	**842**	5108	9423	2186	2090	3482	1531	1533
20	4	32	3620	4329	6799	9101	14867	4204	2250	3584	**1550**	1568
20	8	64		7532		17063	24634	8317	2405	3758	1650	1660
20	16	128		13863		33001	43280	16415	2579	4002	1795	1801
50	1	8	**452**	1824	463	3102	5563	764	1961	3490	1677	1628
50	2	16	**661**	2524	832	5114	8706	1349	2437	3938	1341	1369
50	4	32	2168	4024	4762	9126	14485	2749	2827	4273	**1525**	1539
50	8	64		7061		17110	24886	5480	3079	4457	1781	1789
50	16	128		13038		33092	44273	10868	3301	4624	2056	2062
80	1	8	**494**	1823	553	3101	5479	705	2019	3559	1575	1592
80	2	16	**697**	2522	918	5115	8690	1191	2624	4135	1365	1392
80	4	32	**1618**	3943	3470	9136	14707	2295	3353	4826	1656	1669
80	8	64		6839		17156	25557	4408	3982	5393	2072	2079
80	16	128		12743		33148	45630	8781	4286	5632	2525	2530

Figure 7.4: The Five Fastest exs-Programs

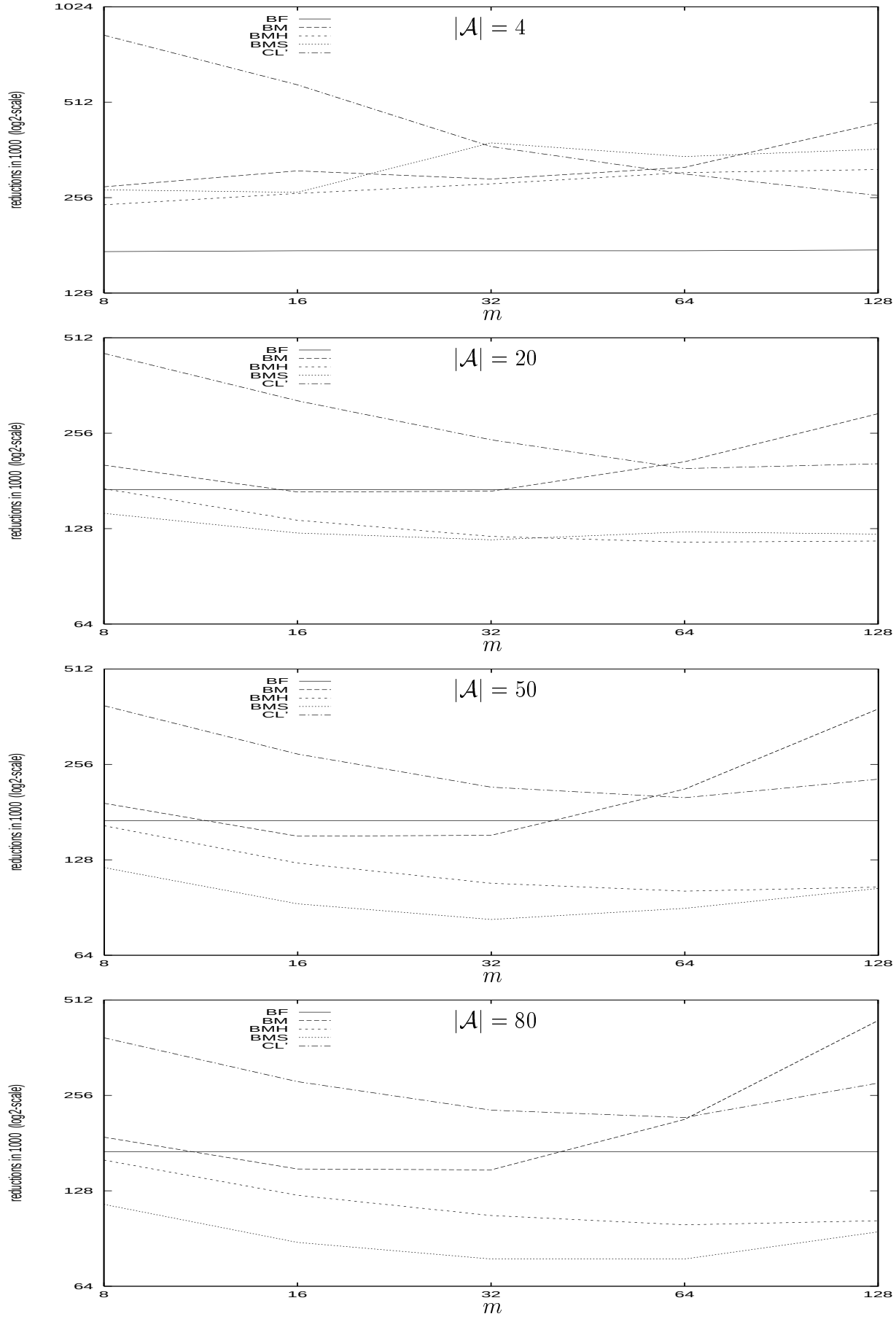


Figure 7.5: The exms-Programs

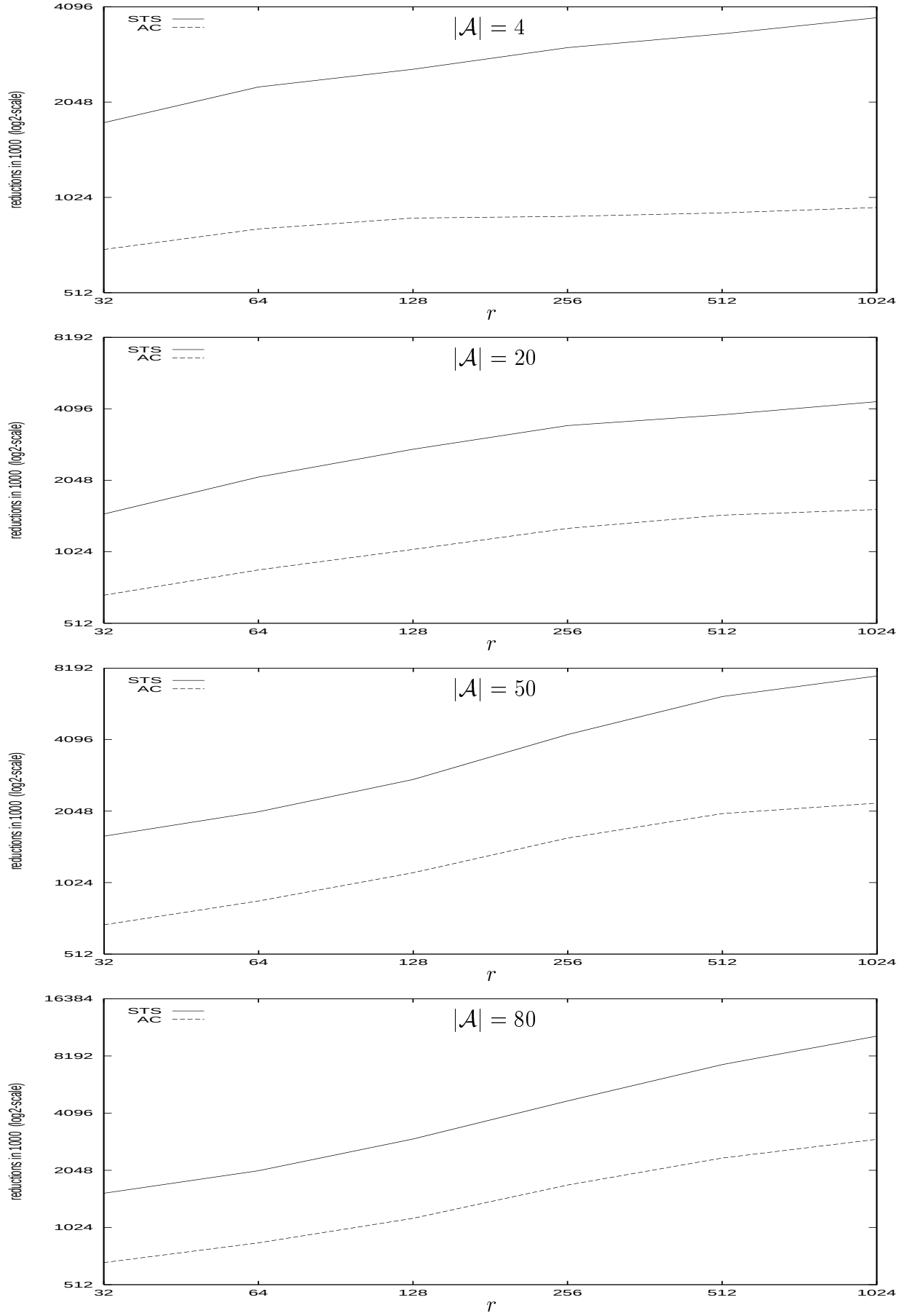


Figure 7.6: First Test Series for the app-Programs ($k = 4$)

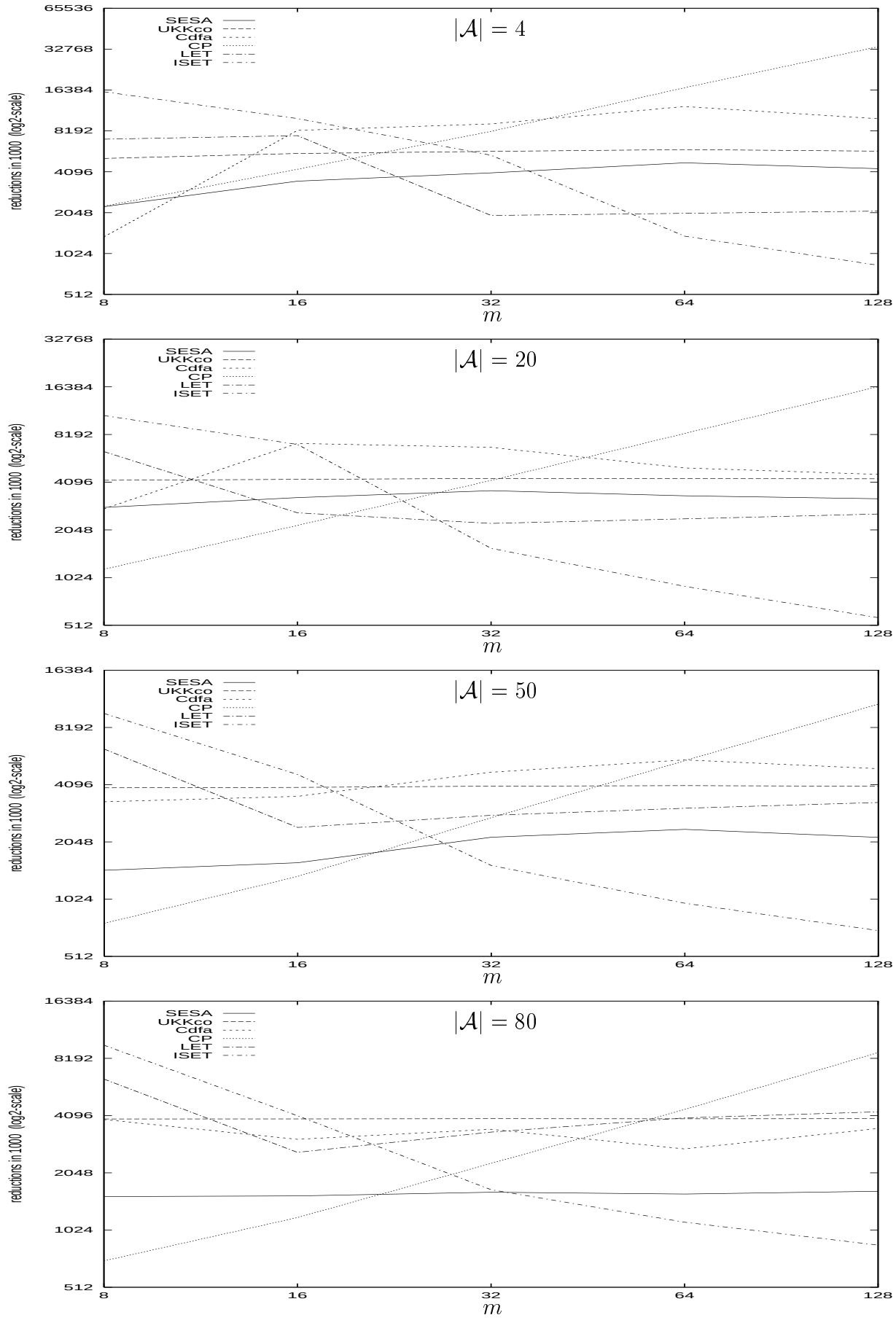


Figure 7.7: Second Test Series for the app-Programs ($m = 10$)

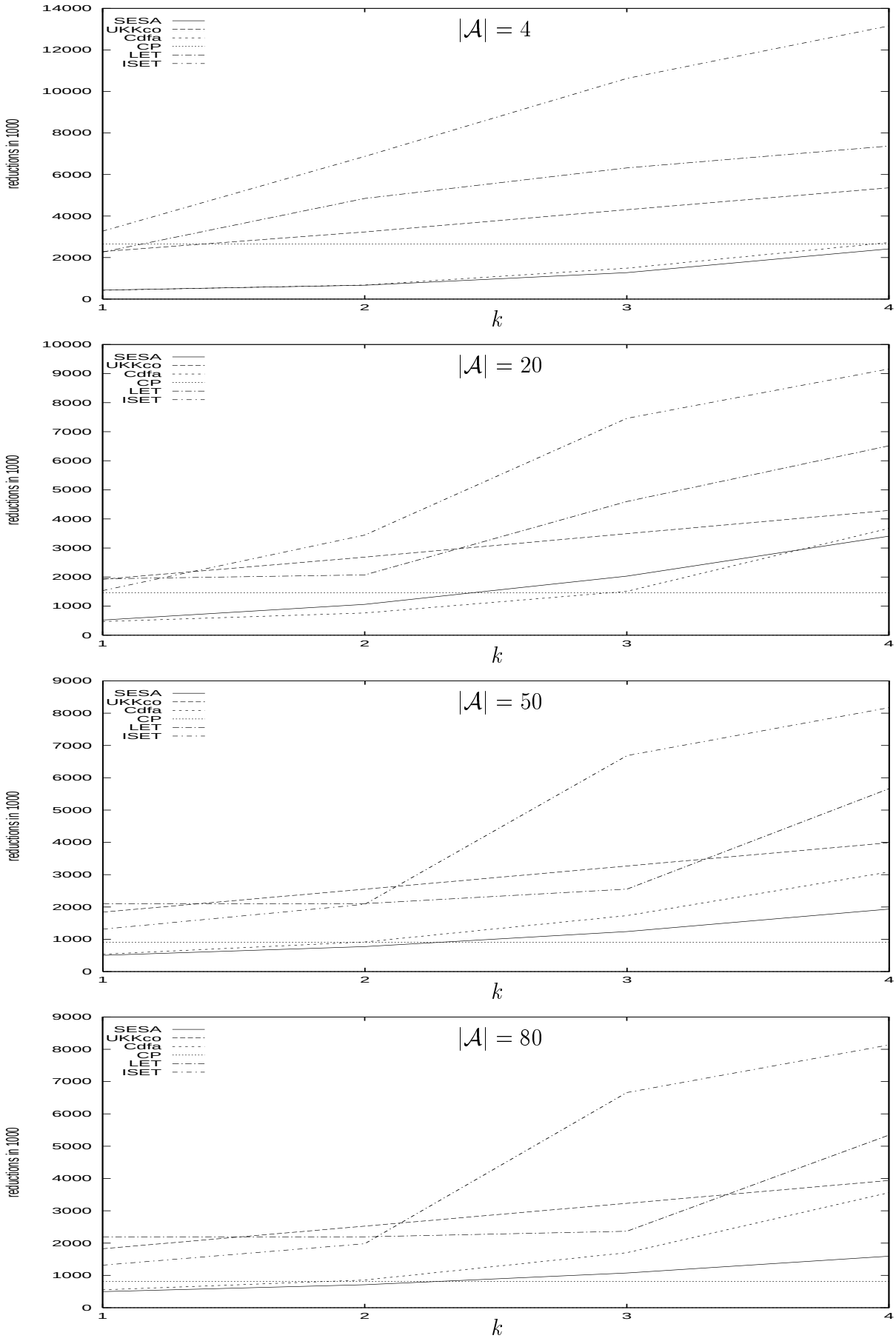
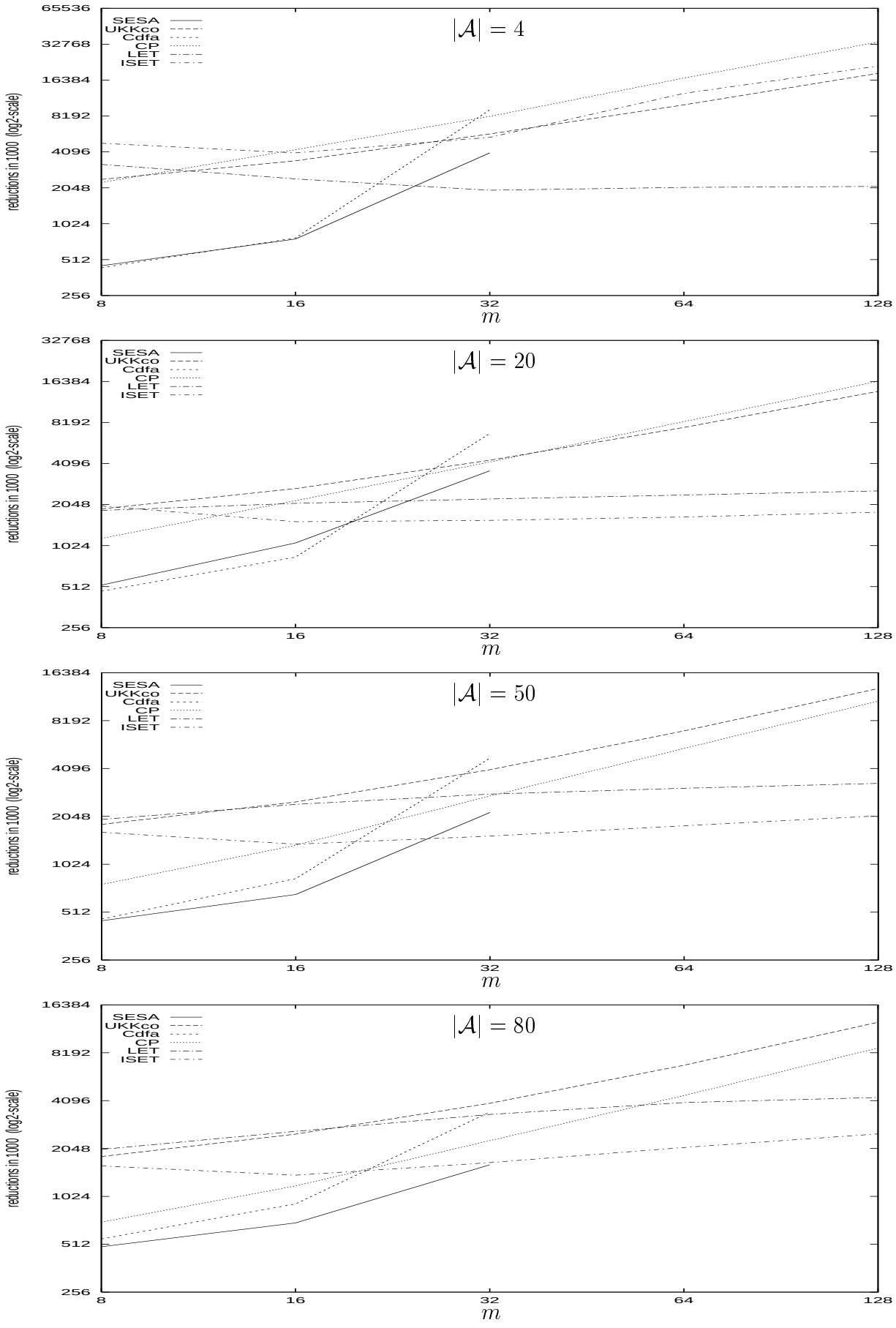


Figure 7.8: Third Test Series for the app-Programs ($k = m/8$)



Chapter 8

Conclusion

Main Findings

In this thesis, we have described how to systematically embed fundamental string algorithms in the functional programming paradigm. Since the programming techniques in the functional and in the imperative world are mostly incompatible, the well-known imperative descriptions of string algorithms were an unsuitable starting point for our purposes. For this reason we have always redeveloped the considered string algorithms from a declarative point of view. In particular, we have elaborated the basic ideas and the properties that are responsible for the correctness and efficiency of the algorithms. The declarative development improved the understanding of the algorithms, revealed some inconsistencies in the literature, and allowed precise and clear correctness proofs. Moreover, it led to robust and efficient functional implementations in a transparent and straightforward way. Finally, as an immediate spin-off, the declarative approach led to the development of some new algorithms:[1]

- Algorithm LOT computes the q-gram distance of two strings of length m and n by a parallel walk of the compact suffix trees, using $\mathcal{O}(m+n)$ space and $\mathcal{O}((|\mathcal{A}|+q)\cdot(m+n))$ time. Algorithm LOT is simpler than previous methods and can easily be implemented in our functional framework.

- Algorithm GST uses the compact suffix tree of the pattern to precompute the functions needed for the Boyer-Moore Algorithm. Algorithm GST runs in $\mathcal{O}(m)$ space and $\mathcal{O}(|\mathcal{A}_p| \cdot m)$ time. It is much simpler than the preprocessing methods presented in the literature.

- Algorithm SESA incrementally constructs a deterministic finite automaton to solve the approximate string searching problem. It runs in $\mathcal{O}(r \cdot (|\mathcal{A}| + m))$ space and $\mathcal{O}(r \cdot |\mathcal{A}| \cdot m + n)$ time where r is the number of states created. Practical experiments show that Algorithm SESA often is faster than previous algorithms for solving the k-differences problem. This is remarkable because it does not exploit any property of the unit cost model, like the competing algorithms do.

[1] Recall the following notations: m is the length of the pattern, n is the length of the input string, $\mathcal{A}$ is the input alphabet, $\mathcal{A}_p \subseteq \mathcal{A}$ is the pattern alphabet, and q is a positive integer.

- By using a more sensitive filter, we have obtained Algorithm ISET, which improves on Chang and Lawler's sublinear expected time algorithm SET. Preliminary experiments show that Algorithm ISET is sometimes 20 percent faster than Algorithm SET.

A main contribution of the thesis is a complete implementation of a variety of string searching and string comparison algorithms. We used the lazy functional language Miranda with two extensions: Extension 1 provided arrays with an efficient creation and lookup function. Extension 2 additionally included an efficient array update function. Building our implementation from individual reusable components, we achieve a higher degree of modularity than is usually seen. It is especially the use of higher order functions, polymorphic types, and lazy evaluation that supports the reusability. Let us recall some important points where we have benefited from these language features:

- We have represented and constructed various forms of $\mathcal{A}^+$-trees in a unified way. The use of higher order functions and polymorphic types made it possible to abstract from the different underlying alphabets, the different degrees of compression, and the application-specific annotations of the nodes.

- The Wagner-Fischer Algorithm for computing the edit distance and optimal alignments occurs in many variants throughout the thesis. We have implemented all these variants in a unified way. By using a higher order function (namely *absnextcol*) and polymorphic types, we were able to abstract from the different ways a new entry in the dynamic programming table is computed from previous table entries.

- We have conveniently realized the search phase of several string searching algorithms which scan an input string character by character, thereby performing state transitions. By using polymorphic types and a higher order function (namely *scanl*), we were able to abstract from the different forms of states and state transition functions.

- The monad of state transformers provides a higher order function (namely *bind*) which is used to encapsulate state based computations so that their external behavior is purely functional. On the basis of this monad, we have developed the first purely functional implementations of linear time suffix tree constructions.

- The four filter algorithms for solving the k-differences problem differ only in the way that interesting subwords are computed. By using a higher order function (namely *appfilter*), we were able to abstract from this difference.

- We have implemented several incremental constructions of suffix trees and deterministic finite automata. Due to the laziness, the incrementality was achieved for free.

- We have implemented a simple method for computing the suffix links of $\mathcal{A}^+$-trees. Due to the laziness, we did not have to concern ourselves with the order in which the suffix links are constructed.

- We have implemented a table specification to enumerate all alignments of two strings. Due to the laziness, we can logically separate the enumeration of the alignments and their selection. In this way, we obtained a very flexible method to compute suboptimal alignments.

- We have represented input strings by lazy lists. In this way, we were able to implement space efficient string searching algorithms which do not require an explicit mechanism for buffering the input string.

Modularity, elegance, and flexibility is, of course, not the only criterion to assess the quality of programs. An important aspect, which has often been neglected in the literature about functional programming (see [Rea89, Hol91]), is the asymptotic efficiency of functional programs. We have not neglected this aspect in the thesis. By using the computational model of outermost graph reduction, we have analyzed the efficiency of our implementations. It was shown that (except for Algorithm DTpp) all functional programs achieve the same asymptotic running time as their imperative counterparts. The analytical results were confirmed by practical measurements. Our efficiency results are remarkable, because it has been questioned that functional programs can generally achieve the same complexity class as imperative programs (see [PMN88]).

In order to obtain efficient functional programs, we have often liberated from implementation techniques developed for imperative languages: For instance, rather than arrays, we used lists to represent strings. Rather than pointers, we used shared subexpressions. Rather than directly updating arrays, we used monadic programming style to guarantee single-threaded arrays. Instead of manipulating a global state, we organized the computation such that the required information is available locally. These and other differences often led to functional programs that did not have much in common with the corresponding imperative programs.

A remarkable feature of our implementation is the rare use of the extensions to Miranda. Extension 2 is only used for implementing linear time suffix tree constructions and for the preprocessing phase of Horspool's and Sunday's exact string searching algorithms. Extension 1 is exclusively used for storing and efficiently looking up preprocessed information about the input alphabet and the pattern. This means that (except for the linear time suffix tree constructions) we utilize only "small" arrays of size $|\mathcal{A}|$ or size m.

Future Research

An important direction of future research is to analyze the new algorithms in more detail. Algorithm LOT and its variation (see page 81) should be compared to Ukkonen's suffix tree based algorithm for computing the q-gram distance. An interesting question is to what extent both algorithms benefit from the lazy suffix tree algorithm. Algorithm GST should be compared to previous preprocessing methods for the Boyer-Moore Algorithm. The comparison should be performed on a practical as well as on a theoretical basis. Algorithm SESA should be analyzed with respect to the number of states and transitions generated if the *SES*-automaton is constructed incrementally. A comparison with the competing algorithm of Sellers should be undertaken using different cost functions. Another important research issue is to compare the different implementation techniques for Algorithm MDC. An interesting question is whether the simplicity of our new online implementation technique leads to an improved running time. Algorithm ILET and ISET should be examined on a theoretical and practical basis. That is, analytical results should be obtained to quantify the threshold value for which the algorithms are linear and "sublinear," respectively. Moreover, the algorithms should be measured with respect to the number of interesting positions computed.

An obvious direction of future research is to apply our string processing machinery to a wider

class of problems on strings. This includes three problems which have interesting applications in molecular biology: the approximate substring matching problem, the approximate overlap problem, and the approximate codon matching problem. For a description of these problems and ideas for their solution, see [CL94]. Generalizations of the approximate string searching problem should be considered as well. Approximate searching of limited regular expressions [WMM92], of network expressions [Mye92b], and of regular expressions [MM89, Mye92a, KM92]. Another important application area is the search for structural patterns, like repeats and palindromes. We are currently developing corresponding search functions which are based on compact suffix trees. An interesting generalization, which is also quite important for molecular biology applications, is the search for approximate structural patterns. Little work has been done in this field (see [LS93, Sch95]).

A wide area of future work will be to perform more experiments on larger data sets. In order to accomplish this, our programs should be translated to the lazy functional languages Clean [PE95] or Haskell [FHPJW92]. Both languages support updatable arrays and provide state-of-the-art compilers for generating native object code. A translation should not be too difficult since in our implementations we have restricted ourselves to those features of Miranda that are (with similar syntax and semantics) also present in Clean and Haskell. Moreover, we have clearly separated our extensions from the basic language features of Miranda.

In our eyes, it is an open question if either Clean or Haskell is the better choice for pragmatic reasons. Clean is quite interesting since it includes a special type system called "uniqueness typing" (see [BS93, SBEP94]) which employs information to deduce whether a data structure is single threaded at a certain moment. By using uniqueness typing, we do not need a monadic programming style to ensure single threadedness. This may improve the readability of functional programs which perform state-based computations. An advantage of Haskell is that its implementations provide advanced profiling facilities (see [RW93, SPJ95]) which is very important in tuning programs. To our knowledge, Clean does not support advanced profiling facilities.

A main goal in the future is to make available our functional implementations as a library for string processing applications. Due to the features we have shown in the thesis, this could make functional programming the technique of choice in a multitude of string processing applications. With improvements in compiler technology for functional languages, the efficiency advantage of imperative programs over functional programs will decrease more and more, so that, in the future, tradition will be the main motivation for using imperative languages when implementing string processing applications.

Appendix A

Implementation of Queues

The polymorphic abstract data type (*queue* α) supports the following functions:

```
emptyqueue::queue α
queueisempty::(queue α)→bool
enqueue::(queue α)→α→(queue α)
dequeue::(queue α)→(α,queue α)
```

emptyqueue returns the empty queue. (*queueisempty* q) checks if q is the empty queue. (*enqueue* q a) adds an element a to the back of q. *dequeue* is applied to a non-empty queue q and returns a pair (a, q') where a is the first element of the front of q and q' is q after removing a from the front. For a formal specification of these operations see [BW88].

The simplest functional implementation represents the queue $(q_1, \ldots, q_l)$ by the list $q = [q_1, \ldots, q_l]$. That is, the beginning of the list acts as the front of the queue and the end of the list acts as the back of the queue. Unfortunately, this representation means that (*enqueue* q a) appends an element to the end of q which requires $\mathcal{O}(|q|)$ time. An alternative solution was suggested by Robert Giegerich [Gie94b]. It represents the queue $(q_1, \ldots, q_l)$ by a function $q :: [\alpha] \to [\alpha]$ such that $q\ [\,] = [q_1, \ldots, q_l]$. The four functions above are defined as follows:

```
emptyqueue = id
queueisempty q = (q [] = [])
enqueue q a = q.(a:)
dequeue q = ((hd.q) [],drop 1.q)
```

It is easily verified that each operation takes constant time. However, in practice this solution proved to be quite slow. For this reason, we apply the implementation technique of Holyer [Hol91]. The idea is to split a queue into two lists. One list contains some elements of the front of the queue, and the other list contains the remaining elements. In particular, the queue $(q_1, \ldots, q_l)$ is represented by a pair (x, y) of type

```
queue α == ([α],[α])
```

such that the following holds:

1. $x +\!\!+ (reverse\ y) = [q_1, \ldots, q_l]$ and
2. $l > 0$ implies $x \neq [\,]$.

x is the front and y is the back of the queue. This representation is not unique. It depends on the history of accesses. The empty queue has an empty front and an empty back. Hence, the first two functions can be defined as follows:

```
emptyqueue::queue α
emptyqueue = ([],[])

queueisempty::(queue α)→bool
queueisempty ([],[]) = True
queueisempty (x,y) = False
```

If q is empty, then *enqueue* adds the element a to the front. If q is not empty, then a is added to the back. This ensures conditions 1 and 2 and takes constant time in both cases.

```
enqueue::(queue α)→α→(queue α)
enqueue ([],[]) a = ([a],[])
enqueue (x,y) a = (x,a:y)
```

dequeue is applied to a non-empty queue q which, by conditions 1 and 2, has a non-empty front. It returns the first element of the front and the remaining queue. If this makes the front empty, the back is reversed and used to replace the front. This ensures conditions 1 and 2.

```
dequeue::(queue α)→(α,queue α)
dequeue ([a],y) = (a,(reverse y,[]))
dequeue (a:x,y) = (a,(x,y))
```

dequeue takes more than constant time when the front contains only one element. However, one can show that *dequeue* takes constant time on the average: Any particular element passed through the queue is removed once from the front and moved once from the back to the front. Removing an element takes constant time. Since $(reverse\ y)$ is evaluated in $\mathcal{O}(|y|)$, the total time spend to move r elements is therefore proportional to r. Hence, moving an element takes constant time on the average.

Appendix B

Predefined Functions

x ++ y appends the lists x and y in $\mathcal{O}(|x|)$ time.

```
(++)::[α]→[α]→[α]
[] ++ y = y
(a:x) ++ y = a:(x ++ y)
```

$x!i$ returns the i-th element of x in $\mathcal{O}(i)$ steps.

```
(!)::[α]→num→α
(a:x)!0 = a
(a:x)!(i+1) = x!i
```

$\#x$ returns the length of x in $\mathcal{O}(|x|)$ steps.

```
(#)::[α]→num
#[] = 0
#(a:x) = 1 + #x
```

const is a function for creating constant valued functions. For instance, (*const* [ ]) is the function that always returns the empty list.

```
const::α→β→α
const x y = x
```

(*drop* i x) returns in $\mathcal{O}(\min\{i, |x|\})$ time the list which remains when the first i elements are removed from x. If $|x| \leq i$, then (*drop* i x = [ ]). Hence, (*drop* 1) is the identity on [ ].

```
drop::num→[α]→[α]
drop (i+1) (a:x) = drop i x
drop i x = x
```

(*dropwhile* f x) drops the longest prefix of x whose elements satisfy f. The running time is proportional to the length of this prefix.

```
dropwhile::(α→bool)→[α]→[α]
dropwhile f [] = []
dropwhile f (a:x) = dropwhile f x, if f a
                  = a:x,           otherwise
```

fst returns the first component of a pair. *snd* returns the second component of a pair. It is defined correspondingly.

```
fst::(α,β)→α
fst (a,b) = a
```

If $x \neq [\,]$, then $(init\ x)$ returns x without its last element. $(init\ [\,])$ is undefined.

```
init::[α]→[α]
init (a:x) = [],       if x = []
           = a:init x, otherwise
```

If $x \neq [\,]$, then $(last\ x)$ returns the last element of x in $\mathcal{O}(|x|)$ steps. $(last\ [\,])$ is undefined.

```
last::[α]→α
last x = x!(#x-1)
```

$(max2\ a\ b)$ returns the maximum of a and b under the built-in ordering $<$. $min2$ returns the minimum. It is defined correspondingly.

```
max2::α→α→α
max2 a b = b, if a < b
         = a, otherwise
```

$(take\ i\ x)$ returns the prefix of x of length i. If $|x| \leq i$, then $(take\ i\ x = x)$. The running time is $\mathcal{O}(\min\{|x|, i\})$.

```
take::num→[α]→[α]
take (i+1) (a:x) = a:take i x
take i x = []
```

$(takewhile\ f\ x)$ takes the longest prefix of x whose elements satisfy f. The running time is proportional to the length of this prefix.

```
takewhile::(α→bool)→[α]→[α]
takewhile f [] = []
takewhile f (a:x) = a:takewhile f x, if f a
                  = [],              otherwise
```

$(zip2\ x\ y)$ returns a list of pairs, formed by pairing corresponding elements of x and y. The running time is $\mathcal{O}(\min\{|x|, |y|\})$.

```
zip2::[α]→[β]→[(α,β)]
zip2 (a:x) (b:y) = (a,b):zip2 x y
zip2 x y = []
```

There are predefined functions $zip3$ and $zip4$ which zip three and four lists to yield a list of triples and quadruples, respectively. They are defined in an analogy to $zip2$.

Bibliography

[AC75] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, **18**:333–340, 1975.

[ACGM92] A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors. *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, Tucson, Arizona, April/May 1992.* Lecture Notes in Computer Science **644**, Springer Verlag, 1992.

[ACGM93] A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors. *Proceedings of the Fourth Annual Symposium on Combinatorial Pattern Matching, Padova, Italy, June 1993.* Lecture Notes in Computer Science **684**, Springer Verlag, 1993.

[AF91] A. Amir and M. Farach. Adaptive Dictionary Matching. In *Proceedings 32nd IEEE Symposium on Foundations of Computer Science*, pages 760–766, 1991.

[AFM92] A. Amir, M. Farach, and Y. Matias. Efficient Randomized Dictionary Matching Algorithms. In [ACGM92], pages 262–275, 1992.

[AG85] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words.* Springer Verlag, 1985.

[AGM+89] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A Basic Local Alignment Search Tool. *J. Mol. Biol.*, **215**:403–410, 1989.

[Aho90] A. Aho. Algorithms for Finding Patterns in Strings. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science, Volume A*, pages 257–300. Elsevier, 1990.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[AHU82] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1982.

[AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, **32**:127–141, 1989.

[AL89] S.F. Altschul and D.J. Lipman. Trees, Stars, and Multiple Biological Sequence Alignment. *SIAM J. Appl. Math.*, **49**(1):197–209, 1989.

[All92] L. Allison. Lazy Dynamic-Programming can be Eager. *Information Processing Letters*, **43**:207–212, 1992.

[AN95] A. Andersson and S. Nilsson. Efficient Implementation of Suffix Trees. *Software — Practice and Experience*, **25**(2):129–141, 1995.

[AP83] A. Apostolico and F.P. Preparata. Optimal Off-line Detection of Repetitions in a String. *Theoretical Computer Science*, **22**:297–315, 1983.

[AP85] A. Apostolico and F.P. Preparata. Structural Properties of the String Statistics Problem. *Journal of Computer and System Sciences*, **31**:394–411, 1985.

[Apo85] A. Apostolico. The Myriad Virtues of Subword Trees. In [AG85], pages 85–96, 1985.

[Apo94] A. Apostolico. Guest Editor's Foreword. *Algorithmica*, **12**(4/5):245–246, 1994.

[AS84] H. Abelson and G.J. Sussman. *The Structure and Interpretation of Computer Programs*. MIT-Press, Cambridge, MA, 1984.

[AS92] A. Apostolico and W. Szpankowski. Self-Alignments in Words and Their Applications. *Journal of Algorithms*, **13**:446–467, 1992.

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1985.

[BBH+85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science*, **40**:31–55, 1985.

[Ber95] D. Berndt. Ähnlichkeitsmaße für Sequenzen basierend auf Teilwortgraphen. Master's Thesis, Technische Fakultät, Universität Bielefeld, 1995.

[BG93] B. Brehme and R. Giegerich. So zählt Miranda. Unpublished Manuscript, Technische Fakultät, Universität Bielefeld, 1993.

[BM77] R.S. Boyer and J.S. Moore. A Fast String Searching Algorithm. *Communications of the ACM*, **20**(10):762–772, 1977.

[Bos88] D.R. Boswell. A Program for Template Matching of Protein Sequences. *CABIOS*, **4**(3):345–350, 1988.

[BRCR94] P. Bieganski, J. Riedl, J.V. Carlis, and E.F. Retzel. Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation. In *Proceedings of the IEEE 27th Hawaii International Conference on System Sciences*, pages 35–44. IEEE Computer Society Press, 1994.

[Bre94] D. Breslauer. Dictionary-Matching on Unbounded Alphabets: Uniform Length Dictionaries. In [Gus94], pages 184–197, 1994.

[BS93] E. Barendsen and S.L Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. Report CSI-R9328, Computing Science Department, University of Nijmegen, 1993.

[BW88] R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[BY89a] R.A. Baeza-Yates. *Efficient Text Searching.* Ph.D. Thesis, Department of Computer Science, University of Waterloo. Also as Research Report CS-89-17, 1989.

[BY89b] R.A. Baeza-Yates. Improved String Matching. *Software — Practice and Experience*, **19**(3):257–271, 1989.

[BY89c] R.A. Baeza-Yates. String Searching Algorithms Revisited. In *Workshop on Algorithms and Data Structures (WADS'89)*, pages 75–96. Lecture Notes in Computer Science **382**, Springer Verlag, 1989.

[BY92] R.A. Baeza-Yates. String Searching Algorithms. In W. Frakes and R.A. Baeza-Yates, editors, *Information Retrieval: Algorithms and Data Structures*, pages 219–240. Prentice Hall, Englewood Cliffs, NJ, 1992.

[BYCG94] R.A. Baeza-Yates, C. Choffrut, and G.H. Gonnet. On Boyer-Moore Automata. *Algorithmica*, **12**(4/5):268–292, 1994.

[BYG92] R.A. Baeza-Yates and G.H. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, **35**(10):74–82, 1992.

[BYP92] R.A. Baeza-Yates and C.H. Perleberg. Fast and Practical Approximate String Matching. In [ACGM92], pages 185–192, 1992.

[BYR90] R.A. Baeza-Yates and M. Régnier. Fast Algorithms for Two Dimensional and Multiple Pattern Matching. In *Proceedings of Second Scandinavian Workshop on Algorithm Theory, SWAT'90*, pages 332–347. Lecture Notes in Computer Science **447**, Springer Verlag, 1990.

[BYR92] R.A. Baeza-Yates and M. Régnier. Average Running Time of Boyer-Moore-Horspool Algorithm. *Theoretical Computer Science*, **92**(1):19–31, 1992.

[CCG+94] M. Crochemore, L. Czumaj, S. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter. Speeding Up Two String-Matching Algorithms. *Algorithmica*, **12**(4/5):247–267, 1994.

[CG89] K.Y. Cockwell and I.G. Giles. Software Tools for Motif and Pattern Scanning: Program Descriptions including a Universal Sequence Reading Algorithm. *CABIOS*, **5**(3):227–232, 1989.

[CHM+86] B. Clift, D. Haussler, R. McConnell, T.D. Schneider, and G.D. Stormo. Sequence Landscapes. *Nucleic Acids Research*, **14**(1):141–158, 1986.

[CL88] H. Carillo and D. Lipman. The Multiple Sequence Alignment Problem in Biology. *SIAM Journal of Applied Mathematics*, **48**(5):1073–1082, 1988.

[CL90] W.I. Chang and E.L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proceedings 31st IEEE Symposium on Foundations of Computer Science*, pages 116–124, 1990.

[CL92] W.I. Chang and J. Lampe. Theoretical and Empirical Comparisons of Approximate String Matching Algorithms. In [ACGM92], pages 175–184, 1992.

[CL94] W.I. Chang and E.L. Lawler. Sublinear Approximate String Matching and Biological Applications (revised version of [CL90]). *Algorithmica*, **12**(4/5):327–344, 1994.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT-Press, Cambridge, MA, 1990.

[CM94] W.I. Chang and T.G. Marr. Approximate String Matching and Local Similarity. In [Gus94], pages 259–273, 1994.

[CMP94] C. Clack, C. Myers, and E. Poon. *Programming with Miranda*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[Col90] R. Cole. Tight Bounds on the Complexity of the Boyer-Moore Pattern Matching Algorithm. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 224–233, 1990.

[Cro86] M. Crochemore. Transducers and Repetitions. *Theoretical Computer Science*, **45**:63–86, 1986.

[Cro88] M. Crochemore. String Matching with Constraints. In *Proceedings of 1988 International Symposium on Mathematical Foundations of Computer Science*, pages 44–58. Lecture Notes in Computer Science **324**, Springer Verlag, 1988.

[CS85] M.T. Chen and J.I. Seiferas. Efficient and Elegant Subword Tree Construction. In [AG85], pages 97–107, 1985.

[CW79] B. Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proceedings ICALP '79*, pages 118–132. Lecture Notes in Computer Science **71**, Springer Verlag, 1979.

[CWM84] Fraser. C., A. Wendt, and E.W. Myers. Analyzing and Compressing Assembly Code. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Volume 19, No. 6*, pages 117–121, 1984.

[DHS84] J. Devereux, P. Haeberli, and O. Smithies. A Comprehensive Set of Sequence Analysis Programs for the VAX. *Nucleic Acids Research*, **12**(1):387–395, 1984.

[EH88] A. Ehrenfeucht and D. Haussler. A New Distance Metric on Strings Computable in Linear Time. *Discrete Applied Mathematics*, **20**:191–203, 1988.

[FG89] E.R. Fiala and D.H. Greene. Data Compression with Finite Windows. *Communications of the ACM*, **32**(4):490–505, 1989.

[FHPJW92] J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Haskell Special Issue. *ACM SIGPLAN Notices*, **27**(5), 1992.

[GBY91] G.H. Gonnet and R.A. Baeza-Yates. Text Algorithms. In *Handbook of Algorithms and Data Structures in Pascal and C*, pages 251–288. Addison-Wesley, Reading, MA, 1991.

[Ger91] German Cancer Research Center and Center for Molecular Biology. *Short Descriptions for the Program Package HUSAR: Heidelberg Unix Sequence Analysis Resources*, 1991.

[GG88] Z. Galil and R. Giancarlo. Data Structures and Algorithms for Approximate String Matching. *Journal of Complexity*, **4**:33–72, 1988.

[GHT84] H. Glaser, C. Hankin, and D. Till. *Principles of Functional Programming.* Prentice Hall, Englewood Cliffs, NJ, 1984.

[Gie92] R. Giegerich. Embedding Sequence Analysis in the Functional Programming Paradigm – A Feasibility Study. Report Nr. 8, Technische Fakultät, Universität Bielefeld, 1992.

[Gie94a] R. Giegerich. Algorithmen auf Sequenzen. Lecture Notes, Technische Fakultät, Universität Bielefeld, 1994.

[Gie94b] R. Giegerich. Personal Communication, 1994.

[GK94] R. Giegerich and S. Kurtz. Suffix Trees in the Functional Programming Paradigm. In *Proceedings of the European Symposium on Programming (ESOP'94)*, pages 225–240. Lecture Notes in Computer Science **788**, Springer Verlag, 1994.

[GK95a] R. Giegerich and S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. *Science of Computer Programming*, **25**(2-3), 1995.

[GK95b] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *accepted for publication in Algorithmica, also available as Report Nr. 94-03, Technische Fakultät, Universität Bielefeld*, 1995.

[GKS95] R. Giegerich, S. Kurtz, and J. Stoye. Lazy Suffix Trees. *In preparation*, 1995.

[Gon92] G.H. Gonnet. A Tutorial Introduction to Computational Biochemistry Using Darwin. Draft Research Report, ETH Zürich, 1992.

[GP89] Z. Galil and K. Park. An Improved Algorithm for Approximate String Matching. In *Proceedings ICALP '89*, pages 394–404. Lecture Notes in Computer Science **372**, Springer Verlag, 1989.

[GP90] Z. Galil and K. Park. An Improved Algorithm for Approximate String Matching, (revised version of [GP89]). *SIAM Journal on Computing*, **19**(6):989–999, 1990.

[Gus94] D. Gusfield, editor. *Proceedings of the Fifth Annual Symposium on Combinatorial Pattern Matching, Asilomar, California, June 1994.* Lecture Notes in Computer Science **807**, Springer Verlag, 1994.

[HD91] A. Hume and Sunday D.M. Fast String Searching. *Software — Practice and Experience*, **21**(11):1221–1248, 1991.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[Hin92] R. Hinze. *Eine Einführung in die Funktionale Programmierung mit Miranda.* Teubner-Verlag, Stuttgart, 1992.

[Hir75] D.S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, **18**:341–343, 1975.

[HL93] P.H. Hartel and K.G. Langendoen. Benchmarking Implementations of Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, June 1993*, pages 341–349. ACM Press, New York, NY, 1993.

[HO80] G. Huet and D.C. Oppen. Equations and Rewrite Rules: A Survey. In Book, R.V., editor, *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

[Hol91] I. Holyer. *Functional Programming with Miranda.* Pitman, 1991.

[Hor80] R.N. Horspool. Practical Fast Searching in Strings. *Software — Practice and Experience*, **10**(6):501–506, 1980.

[HT84] D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, **13**:338–355, 1984.

[Hud93] P. Hudak. ACM SIGPLAN Workshop on State in Programming Languages. Technical report, Yale University, New Haven, Department of Computer Science, 1993.

[Hug89] J. Hughes. Why Functional Programming matters. *The Computer Journal*, **32**(2):98–107, 1989.

[Hut92] G. Hutton. Higher Order Functions for Parsing. *Journal of Functional Programming*, **3**(2):323–343, 1992.

[IS92] R.M. Idury and A.A. Schäffer. Dynamic Dictionary Matching with Failure Functions. In [ACGM92], pages 276–287, 1992.

[Jon94] M.P. Jones. The Implementation of the Gofer Functional Programming System. Report, YALEU/DCS/RR-1030, Yale University, New Haven, Department of Computer Science, 1994.

[JTU91] P. Jokinen, J. Tarhio, and E. Ukkonen. A Comparison of Approximate String Matching Algorithms. Technical Report A-1991-7, Department of Computer Science, University of Helsinki, 1991.

[JU91] E. Jokinen and E. Ukkonen. Two Algorithms for Approximate String Matching in Static Texts. In *Proceedings of 16th International Symposium on Mathematical Foundations of Computer Science*, pages 240–248. Lecture Notes in Computer Science **520**, Springer Verlag, 1991.

[KBG87] M. Kempf, R. Bayer, and U. Güntzer. Time Optimal Left To Right Construction of Position Trees. *Acta Informatica*, **24**:461–474, 1987.

[KM92] J.R. Knight and E.W. Myers. Approximate Regular Expression Pattern Matching with Concave Gap Penalties. In [ACGM92], pages 67–78, 1992.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, **6**(2):323–350, 1977.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[Kos94] S.R. Kosaraju. Computation of Squares in a String. In [Gus94], pages 146–150, 1994.

[KR87] R.M. Karp and M.O. Rabin. Efficient Randomized Pattern Matching Algorithms. *IBM Journal of Research and Development*, **31**(2):249–260, 1987.

[Kra95] A. Krause. Realisierung von Zustandskonzepten in Funktionalen Programmiersprachen am Beispiel von Linearen Suffixbaum-Konstruktionen. Master's Thesis (in preparation), Technische Fakultät, Universität Bielefeld, 1995.

[KS83] J.B. Kruskal and D. Sankoff. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, 1983.

[KST92a] J.Y. Kim and J. Shawe-Taylor. An Approximate String-Matching Algorithm. *Theoretical Computer Science*, **92**(1):107–117, 1992.

[KST92b] J.Y. Kim and J. Shawe-Taylor. Fast Multiple Keyword Searching. In [ACGM92], pages 41–51, 1992.

[LO94] A. López-Ortiz. Linear Pattern Matching of Repeated Substrings. *ACM SIGACT NEWS*, **25**(3):114–121, 1994.

[LP85] D.J. Lipman and W.R. Pearson. Rapid and Sensitive Protein Similarity Search. *Science*, **227**:1435–1441, 1985.

[LPJ94] J. Launchbury and S.L. Peyton Jones. Lazy Functional State Threads. In *ACM Conference on Programming Language Design and Implementation, Orlando, FL, June 1994*, 1994.

[LS92] J. Launchbury and P.M. Sansom, editors. *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer Verlag, 1992.

[LS93] G.M. Landau and J.P. Schmidt. An Algorithm for Approximate Tandem Repeats. In [ACGM93], pages 120–133, 1993.

[LV86] G.M. Landau and U. Vishkin. Efficient String Matching with k mismatches. *Theoretical Computer Science*, **43**:239–249, 1986.

[LV88] G.M. Landau and U. Vishkin. Fast String Matching with k Differences. *Journal of Computer and System Sciences*, **37**:63–78, 1988.

[LV89] G.M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms*, **10**:157–169, 1989.

[Mac90] B.J. MacLennan. *Functional Programming: Practice and Theory.* Addison-Wesley, Reading, MA, 1990.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, **3**(4):184–195, 1960.

[McC76] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.

[Meh91] G. Mehldau. A Pattern Matching System for Biosequences. Ph.D. Thesis, available as Technical Report TR 91-21, University of Arizona, Tucson, Department of Computer Science, 1991.

[MLC91] F. Major, G. Lapalme, and R. Cedergren. Domain Generating Functions for Solving Constraint Satisfaction Problems. *Journal of Functional Programming*, **1**(2):213–227, 1991.

[MM88] E.W. Myers and W. Miller. Sequence Comparison with Concave Weighting Functions. *Bulletin of Mathematical Biology*, **50**(2):97–120, 1988.

[MM89] E.W. Myers and W. Miller. Approximate Matching of Regular Expressions. *Bulletin of Mathematical Biology*, **51**(1):5–37, 1989.

[MM90] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[MM93a] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches (revised version of [MM90]). *SIAM Journal on Computing*, **22**(5):935–948, 1993.

[MM93b] G. Mehldau and E.W. Myers. A System for Pattern Matching Applications on Biosequences. *CABIOS*, **9**(3):299–314, 1993.

[MP80] U. Masek and M.S. Paterson. A Faster Algorithm for Computing String-Edit Distances. *Journal of Computer and Systems Sciences*, **20**(1):785–807, 1980.

[MR80] M.E. Majster and A. Reiser. Efficient On-line Construction and Correction of Position Trees. *SIAM Journal on Computing*, **9**(4):785–807, 1980.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT-Press, Cambridge, MA, 1990.

[MW93] U. Manber and S. Wu. GLIMPSE: A Tool to Search Through Entire File Systems. Technical Report TR 93-34, University of Arizona, Tucson, Department of Computer Science, 1993.

[Mye86] E.W. Myers. An $\mathcal{O}(ND)$ Differences Algorithm. *Algorithmica*, **2**(1):251–266, 1986.

[Mye91] E.W. Myers. An Overview of Sequence Comparison Algorithms in Molecular Biology. Technical Report TR 91-29, University of Arizona, Tucson, Department of Computer Science, 1991.

[Mye92a] E.W. Myers. A Four-Russians Algorithm for Regular Expression Matching. *Communications of the ACM*, **39**:430–448, 1992.

[Mye92b] E.W. Myers. Approximate Matching of Network Expressions with Spacers. Technical Report TR 92-5, University of Arizona, Tucson, Department of Computer Science, 1992.

[Mye94a] E.W. Myers. A Sublinear Algorithm for Approximate Keyword Searching. *Algorithmica*, **12**(4/5):345–374, 1994.

[Mye94b] E.W. Myers. Algorithmic Advances for Searching Biosequence Databases. In S. Suhai, editor, *Computational Methods in Genome Research*. to appear in Plenum Press, 1994.

[Mye95a] E.W. Myers. Guest Editor's Foreword. *Algorithmica*, **13**(1/2):1–6, 1995.

[Mye95b] E.W. Myers. Incremental Alignment Algorithms and their Application. *to appear in: SIAM Journal on Computing*, 1995.

[NB93] D. Naor and D. Brutlag. On Suboptimal Alignments of Biological Sequences. In [ACGM93], pages 179–196, 1993.

[NS93] E. Nöcker and S. Smetsers. Partially Strict Non-Recursive Data Types. *Journal of Functional Programming*, **2**(3):191–215, 1993.

[NW70] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins. *Journal of Molecular Biology*, **48**:443–453, 1970.

[OM88] O. Owolabi and D.R. McGregor. Fast Approximate String Matching. *Software — Practice and Experience*, **18**:387–393, 1988.

[PE95] R. Plasmeijer and M. van Eekelen. *Concurrent Clean Language Reference Manual*. Computing Science Department, University of Nijmegen, 1995.

[Per90] D. Perrin. Finite Automata. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science, Volume B*, pages 1–57. Elsevier, 1990.

[Pir92] K. Pirklbauer. A Study of Pattern Matching Algorithms. *Structured Programming*, **13**:89–98, 1992.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[PJ89] S.L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, **32**(2):175–186, 1989.

[PJL91] S.L. Peyton Jones and D. Lester. *Implementing Functional Languages*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[PJW91] S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages, Charlston, SC, January 1993*, pages 71–84, 1991.

[PMN88] C.G. Ponder, P.C. McGeer, and A.P. Ng. Are Applicative Languages Inefficient? *SIGPLAN Notices*, **6**:135–139, 1988.

[Pow89] P.A. Powell. Molecular Biological Applications of the Suffix Tree Index Structure for String Queries. Technical Report TR-89-79, Computer Science Department, University of Minnesota, Minneapolis, MN, 1989.

[PW95] P.A. Pevzner and M.S. Waterman. Multiple Filtration and Approximate Pattern Matching. *Algorithmica*, **13**(1/2):135–154, 1995.

[RC86] J. Rees and W. Clinger. The Revised^3 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, **21**(12):37–79, 1986.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

[Rod82] M. Rodeh. A Fast Test for Unique Decipherability Based on Suffix Trees. *IEEE Transactions on Information Theory*, **28**(4):648–651, 1982.

[RPE81] M. Rodeh, V.R. Pratt, and S. Even. Linear Algorithms for Data Compression via String Matching. *Journal of the ACM*, **28**(1):16–24, 1981.

[RW93] C. Runciman and D. Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, **3**(2):217–245, 1993.

[Ryt80] W. Rytter. A Correct Preprocessing Algorithm for Boyer-Moore String-Searching. *SIAM Journal on Computing*, **9**:509–512, 1980.

[San95] T. Sander. Lazy Scanner Generierung. Master's Thesis, Technische Fakultät, Universität Bielefeld, 1995.

[SBEP94] S. Smetsers, E. Barendsen, M. van Eekelen, and R. Plasmeijer. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In H.J. Schneider and H. Ehrig, editors, *Proceedings of the Workshop on Graph Transformations in Computer Science*. Lecture Notes in Computer Science **776**, Springer Verlag, 1994.

[Sch85] D.A. Schmidt. Detecting Global Variables in Denotational Specifications. *TOPLAS*, **7**:299–310, 1985.

[Sch88] R. Schaback. On the Expected Sublinearity of the Boyer-Moore Algorithm. *SIAM Journal on Computing*, **17**(4):648–658, 1988.

[Sch95] J.P. Schmidt. All Shortest Paths in Weighted Graphs and its Application to Finding All Approximate Repeats in Strings. Technical Report, Department of Computer Science, Polytechnic University, Brooklyn, NY 11201, 1995.

[Sea89] D.B. Searls. Investigating the Linguistics of DNA with Definite Clause Grammars. In *Proceedings of the North American Conference on Logic Programming*, pages 189–208. MIT-Press, 1989.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 1988.

[Sel80] P.H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, **1**:359–373, 1980.

[Smi82] G. de V. Smit. A Comparison of Three String Matching Algorithms. *Software — Practice and Experience*, **12**:57–66, 1982.

[Smi88] R. Smith. A Finite State Machine Algorithm for Finding Restriction Sites and other Pattern Matching Applications. *CABIOS*, **4**(4):459–465, 1988.

[Smi91] P.D. Smith. Experiments with Very Fast String Substring Search Algorithms. *Software — Practice and Experience*, **21**(10):1065–1074, 1991.

[SPJ95] P.M. Sansom and S.L. Peyton Jones. Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, San Francisco, CA, January 1995*, pages 355–366, 1995.

[SR92] P. Sanders and C. Runciman. LZW Text Compression in Haskell. In [LS92], pages 215–226, 1992.

[Sta88] R. Staden. Methods to Define and Locate Patterns of Motifs in Sequences. *CABIOS*, **4**(1):53–60, 1988.

[Ste91] M.J.E. Sternberg. PROMOT: A Fortran Program to Scan Protein Sequences Against a Library of Known Motifs. *CABIOS*, **7**(2):257–260, 1991.

[Ste94] G.A. Stephen. *String Searching Algorithms*. World Scientific Publishing, 1994.

[Sto77] J.E. Stoy. *The Scott-Strachey Approach to Programming Language Theory*. MIT-Press, Cambridge, MA, 1977.

[Sto95] J. Stoye. Affixbäume. Master's Thesis, Technische Fakultät, Universität Bielefeld, 1995.

[Sun90] D.M. Sunday. A Very Fast Substring Searching Algorithm. *Communications of the ACM*, **33**(8):132–142, 1990.

[SV88] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors. *SIAM Journal on Computing*, **17**(6):1253–1263, 1988.

[Thi94] P. Thiemann. *Grundlagen der Funktionalen Programmierung*. Teubner-Verlag, Stuttgart, 1994.

[TU93] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore String Matching. *SIAM Journal on Computing*, **22**(2):243–260, 1993.

[Tur79] D.A. Turner. A New Implementation Technique for Applicative Languages. *Software — Practice and Experience*, **9**:31–49, 1979.

[Tur85] D.A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *Proceeding IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, pages 1–16. Lecture Notes in Computer Science **201**, Springer Verlag, 1985.

[Tur86] D.A. Turner. An Overview of Miranda. *SIGPLAN Notices*, **21**(12), 1986.

[Tur89] D.A. Turner. *Miranda System Manual.* Research Software Ltd, 23 St Augustines Road, Canterbury, Kent CT1, 1989.

[Ukk85a] E. Ukkonen. Algorithms for Approximate String Matching. *Information and Control*, **64**:100–118, 1985.

[Ukk85b] E. Ukkonen. Finding Approximate Patterns in Strings. *Journal of Algorithms*, **6**:132–137, 1985.

[Ukk92a] E. Ukkonen. Approximate String-Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, **92**(1):191–211, 1992.

[Ukk92b] E. Ukkonen. Constructing Suffix Trees On-line in Linear Time. *Algorithms, Software, Architecture. J.v.Leeuwen (Ed.), Information Processing 92, Volume I*, pages 484–492, 1992.

[Ukk93a] E. Ukkonen. Approximate String-Matching over Suffix Trees. In [ACGM93], pages 229–242, 1993.

[Ukk93b] E. Ukkonen. On-line Construction of Suffix-Trees (revised version of [Ukk92b]). to appear in: *Algorithmica,* also available as Report, A-1993-1, Department of Computer Science, University of Helsinki, Finland, 1993.

[Ula72] S.M. Ulam. Some Combinatorial Problems Studied Experimentally on Computing Machines. In Zaremba, S.K., editor, *Applications of Number Theory to Numerical Analysis*, pages 1–3. Academic Press, 1972.

[UW93] E. Ukkonen and D. Wood. Approximate String Matching With Suffix Automata. *Algorithmica*, **10**:353–364, 1993.

[vdW89] J. van der Woude. Playing with Patterns, Searching for Strings. *Science of Computer Programming*, **12**(3):177–190, 1989.

[Wad85] P. Wadler. How to Replace Failure by List of Successes. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Lecture Notes in Computer Science **201**, Springer Verlag, 1985.

[Wad90] P. Wadler. Comprehending Monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[Wad92a] P. Wadler. Monads for Functional Programming. In M. Broy, editor, *Program Design Calculi, Proceedings of the International Summer School 1992*. Springer Verlag, NATO ASI series, Series F: Computer and System Sciences, Heidelberg, 1992.

[Wad92b] P. Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium of Programming Languages*, pages 1–14, 1992.

[Wat89a] M.S. Waterman. *Mathematical Methods for DNA Sequences*. CRC-Press, Boca Raton, FL, 1989.

[Wat89b] M.S. Waterman. Sequence Alignments. In [Wat89a], pages 53–92. 1989.

[Wei73] P. Weiner. Linear Pattern Matching Algorithms. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[WF74] R.A. Wagner and M.J. Fischer. The String to String Correction Problem. *Journal of the ACM*, **21**(1):168–173, 1974.

[Wik87] Å. Wikström. *Functional Programming Using Standard ML*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[WM92] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, **10**(35):83–91, 1992.

[WM94] S. Wu and U. Manber. A Fast Algorithm for Multi Pattern Matching. Technical Report TR 94-17, University of Arizona, Tucson, Department of Computer Science, 1994.

[WMM91] S. Wu, U. Manber, and E.W. Myers. Improving the Running Time for some String-Matching Problems. Technical Report TR 91-20, University of Arizona, Tucson, Department of Computer Science, 1991.

[WMM92] S. Wu, U. Manber, and E.W. Myers. A Sub-quadratic Algorithm for Approximate Limited Expression Matching. Technical Report TR 92-36, University of Arizona, Tucson, Department of Computer Science, 1992.

[ZD95] J. Zobel and P. Dart. Finding Approximate Matches in Large Lexicons. *Software — Practice and Experience*, **25**(3):331–345, 1995.

Index

Bisher erschienene Reports an der Technischen Fakultät
Stand: December 18, 1995

94–01 Modular Properties of Composable Term Rewriting Systems
(Enno Ohlebusch)

94–02 Analysis and Applications of the Direct Cascade Architecture
(Enno Littmann and Helge Ritter)

94–03 From Ukkonen to McCreight and Weiner: A Unifying View
of Linear-Time Suffix Tree Construction
(Robert Giegerich and Stefan Kurtz)

94–04 Die Verwendung unscharfer Maße zur Korrespondenz Analyse
in Stereo Farbbildern
(Alois Knoll and Andrè Wolfram)

94–05 Searching Correspondences in Colour Stereo Images
– Recent Results Using the Fuzzy Integral
(Alois Knoll and Andrè Wolfram)

94–06 A Basic Semantics for Computer Arithmetic
(M. Freericks, A. Fauth, A. Knoll)

94–07 Reverse Restructuring: Another Method of Solving
Algebraic Equations
(Bernd Bütow and Stephan Thesing)

95–01 PaNaMa User Manual V1.3
(Bernd Bütow and Stephan Thesing)

95–02 Computer Based Training-Software:
ein interaktiver Sequenzierkurs
(Frank Meier, Garrit Skrock, Robert Giegerich)

95–03 Fundamental Algorithms for a Declarative Pattern Matching System
(Stefan Kurtz)